DTIC FILE COPY

(2)

AD-A223 921

DTIC ELECTE JUL 13 1990 S B D

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE				Form Approved OMB No. 0704-0188
1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED		1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY		3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release: distribution unlimited.		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE				
4. PERFORMING ORGANIZATION REPORT NUMBER(S)		5. MONITORING ORGANIZATION REPORT NUMBER(S)		
6a. NAME OF PERFORMING ORGANIZATION: YALE UNIVERSITY	6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION: Cognitive Science (Code 1142 CS) Office of Naval Research		
6c. ADDRESS (City, State, and ZIP Code): Dept. of Computer Science New Haven, CT 06520		7b. ADDRESS (City, State, and ZIP Code): 800 N. Quincy St. Arlington, VA 22217		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Office of Naval Research	8b. OFFICE SYMBOL (If applicable): Code 1142CS	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER: N00014-82-K-0714		
8c. ADDRESS (City, State, and ZIP Code): 800 N. Quincy Street Arlington, VA 22217-5000		10. SOURCE OF FUNDING NUMBERS		
		PROGRAM ELEMENT NO.: 61153N	PROJECT NO.: RR04206	TASK NO: RR04206C
				WORK UNIT ACCESSION NO.: 4420007

11. TITLE (Include Security Classification)

STRATEGIES FOR TUTORING MULTIPLE BUGS

12. PERSONAL AUTHOR(S): DAVID CARL LITTMAN

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final Report	FROM 8/15/85 TO 11/15/88	1990 May 18	254

16. SUPPLEMENTARY NOTATION

— fr. back

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
05	08		Cognitive Psychology; Intelligent Tutoring; Automated Diagnostic Inference; Computer Progamming; and Instruction
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report investigated a problem in the field of Intelligent Tutoring Systems that has not been effectively addressed before. That problem is how to help students who need help with more than one error, or bug. For example, novice students who write computer programs rarely make a single bug. Rather, they come to their tutors with five, six, seven, or more bugs and they need help with all the bugs.

Empirical observations of experienced human tutors show that they do not just jump in and start tutoring the first bug in the program. Instead, tutors formulate a tutorial plan for helping the student. Creating the tutorial plan requires the tutor to answer the following five tutorial planning decisions about each bug:

TPD1: Should the bug be tutored?

TPD2: Which other bugs should the bug be tutored with?

TPD3: What tutorial objectives does the tutor want to achieve?

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL: Dr. Susan E. Chipman	22b. TELEPHONE (Include Area Code): 202-696-4318 — 22c. OFFICE SYMBOL: ONR, Code 1142CS

DD Form 1473, JUN 86 — Previous editions are obsolete. S/N 0102-LF-014-6603 — SECURITY CLASSIFICATION OF THIS PAGE: Unclassified

Edition.

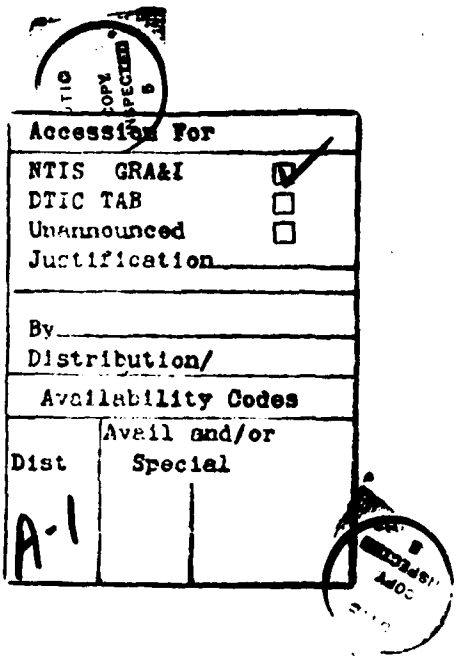
90 07 12 014

(Continued on reverse)

Cont.

TPD4: When should the bug be tutored?
TPD5: What intervention strategies should be used to tutor the bug?

Making all five Tutorial Planning Decisions cannot be done simply by generating all the possible tutorial plans and choosing the best: There are literally hundreds of millions of possible plans for the same set of seven bugs.
Previous work on multiple bugs has not considered problem solving situations in which students make more than two or three bugs. The approaches to multiple bugs developed for simple situations cannot work once the student makes more than three or four bugs.
This research studied the problem of multiple bugs. Starting from empirical observations of experienced human tutors, a model was developed of the knowledge required to answer the five tutorial planning questions. The model is implemented as a computer program, TP, that can develop tutorial plans for multiple bugs that are as good as the tutorial plans generated by experienced human tutors. Keywords: → FLD 18



Abstract

Strategies for Tutoring Multiple Bugs

David Carl Littman
Yale University
1990

The dissertation "Strategies for Tutoring Multiple Bugs" investigates a problem that has not been effectively addressed before. That problem is how to help students who need help with more than one error, or bug. For example, novice students who write computer programs rarely make a single bug. Rather, they come to their tutors with five, six, seven, or more bugs and they need help with *all* the bugs.

Empirical observations of experienced human tutors show that they do not just jump in and start tutoring the first bug in the student's program. Instead, tutors formulate a *tutorial plan* for helping the student. Creating a tutorial plan requires the tutor to answer the following five tutorial planning decisions about each bug:

TPD1. Should the bug be tutored?

TPD2. Which other bugs should the bug be tutored with?

TPD3. What tutorial objectives does the tutor want to achieve?

TPD4. When should the bug be tutored?

TPD5. What intervention strategies should be used to tutor the bug?

Making the five Tutorial Planning Decisions cannot be done by simply generating all the possible tutorial plans and choosing the best: There are literally hundreds of millions of possible plans for the same set of seven bugs.

Previous work on multiple bugs has not considered problem solving situations in which students make more than two or three bugs. The approaches to multiple bugs developed for simple situations cannot work once the student makes more than three or four bugs.

"Strategies for Tutoring Multiple Bugs" explores the problem of tutoring multiple bugs. Starting from empirical observations of experienced human tutors, a model is developed of the knowledge required to answer the five tutorial planning questions. The model is implemented as a computer program, TP, that can develop tutorial plans for multiple bugs.

Sponsored by Cognitive Science Program
Office of Naval Research
Contract # N00014-82-K-0714

Approved for Public Release
Distribution Unlimited

Reproduction in Part or
Whole is Permitted for
any Purpose of the
United States Government

Strategies for Tutoring Multiple Bugs

A Dissertation

Presented to the Faculty of the Graduate School

of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy

by

David Carl Littman

December, 1990

© Copyright by David Carl Littman, 1990
ALL RIGHTS RESERVED

For Ulric Neisser

Acknowledgements

My path to, and through, the period of my life at Yale owes much to many and it is here that I want to acknowledge the influences and support of many different people in several different categories. That is what I intend to do here.

First, there are the two people that have made it possible for me to fulfill a longtime dream --- that of becoming competent in the discipline of Artificial Intelligence. Roger Schank let me come to Yale and watch him think. He also gave me very useful advice at two crucial junctures in my graduate career and for that I give him my deep thanks.

My advisor, Elliot Soloway also made it possible for me to come to Yale but he did much more through the six years that he supported me financially, intellectually, and emotionally. Elliot's motto "The more you succeed the more I succeed" is a deep reflection of his inner self. Elliot's high standards and personal power forced me to do what I had never done before: think and write clearly. Elliot is truly responsible for any successes I have in the future and the few words that appear in any acknowledgement could never do him justice.

Finally, Bob Hodapp, the third member on my committee, and friend, jumped in at the last moment to insure at least a modicum of adherence to the rules of good psychological research. It's not that I don't know the rules, it's just that Bob made sure that I followed them. In addition, his comments at our meetings to discuss drafts of my dissertation were invaluable.

Second, there are several graduate students without whom I would not have survived. First and foremost among them is Jim Spohrer. Jim's help with my dissertation made it possible. His nonthreatening style of criticism and his clear, concrete advice made the difference for me between a dissertation that was just okay and one that I am very pleased to have written. Whenever I worried that my work was unimportant, Jim helped me see that there definitely was a pony in there. His loan of a shovel and his extremely good directions made the path to the pony quite direct. Thanks, Jim. Jim Firby was the one who learned me to program. One very late night he was helping me with an assignment in our compilers class. He looked at some of my code and said (I quote) "You don't know how to program, do you?" To which I replied "No." and to which he replied "Well, let's get started." The proof of Jim's teaching skill, and tenacity, lies in the fact that I passed my programming courses. Other graduate students provided a never-ending source of ideas, entertainment, and support. .These include classmates Brad Alpert, Steve Hanks, Alex Kass, David Leake, Chris Owens and, especially, Larry Hunter. It is not possible to put into words how much I enjoyed thinking and doing with Larry: I simply note that Larry is a major attraction of the geographical region where I will be the next few years. As well as students in my class, there were those who had been at Yale longer than I (Natalie Dehn and Stan Letovsky) as well as several students who arrived after me (Rob Farrell and especially Ashwin Ram, who served as the model for the prototype of my tutorial planner).

Third, my instructors at Yale were, without exception, superb teachers. Of course, Drew McDermott and Larry Birnbaum taught me much about Artificial Intelligence, as did Gene Charniak when he visited from Brown. But Dana Angluin showed me that the theory that lies behind computer science can be fascinating and accessible, and, even more, enjoyable -- especially NP-Completeness, my favorite topic in Theory of Computation. Paul Hudak, a terrific teacher, taught a course about compilers that I really enjoyed. I never thought I would, but I did and that is due to entirely to Paul's skill. David Gelernter's course on operating systems was actually *fun* ! His dry sense of humor was a pleasant departure from the seriousness that seems to pervade Computer Science. I am pleased to have learned from these people. I would also like to thank Alan Perlis for many very enlightening conversations.

Fourth, the two or three generations of staff in the support facility deserve thanks. Not only did I not know how to program when I arrived at Yale, I had no idea how to use computers. I think I must have messed up more than anyone thought possible, but Miriam Putterman, John Philbin, Dave Teodosio, Rob Carey, Morrow Long, and Lamarr Ledbetter saw me through with smiles on their faces -- most of the time.

Fifth, I would like to thank Bob Dunne and Martin Schultz who made the final few months of my stay at Yale painless. I never had to steal a single Xerox copy and for that I thank them. As well, Eric Mjolsness and Padmanaban Anandan lent me a monitor for my MAC II at a crucial time. That made my life bearable for the final push.

Susan Chipman, of the Office of Naval Research, supported virtually my whole graduate career and for that she has my thanks.

Sixth, the secretaries who copied my papers, made sure that I got to Oshkosh, and back when I was supposed to and always had a good word all get a thank you. Donna Alchimio (Mauri) was absolutely wonderful. She NEVER made a mistake. Nor did Judy Terrell, Connie Maher, Julie Teodosio. Michella Schubert, the graduate secretary, was always fun to talk with. Linda Bourne and I spend many pleasant moments together discussing computer scientists and life.

Seventh, many people at Yale were fun to talk to. Pat Cunningham, Kathryn Fontwit, and many others made it possible for me to do what I like best: think about almost everything there is to think about.

Eighth, there are the various and sundry people and pets who have passed through my life. If I left you out, it is because I forgot -- not because I didn't appreciate you. Pauline Pap, one of the niftiest people I know, made me feel good because she liked sharing theater, problems, and joys with me. Becky Anderson was, and is, a good friend with whom I share several common interests. Christiaan Karman and Herman Lenferink made Holland a great place to visit. I always enjoyed my discussions about AI, computer science, and the universe with Bill Clancey. The time I spent on the phone talking about science with Lisa Neal was always profitable and enjoyable. And without Rocky and his now-dead mother, The Mom, I would have had a very different, much less happy, emotional life these last 17

years. I only wish that they understood how much they have meant to me.

Finally, I want to acknowledge the person who has had the greatest intellectual influence on me. When I read the last chapter of his first book I said "I absolutely must go study with him". And I did. I took more from him intellectually than I ever returned and I can only say that, to the extent that I am a good cognitive psychologist, he is responsible. I hope that this, second, dissertation expiates some of the pain I caused him during the creation of the first. If I see any of the reality of cognition, he is responsible. Thus, it is to this man, Dick Neisser, that I dedicate my dissertation. He may not like it, but it is for him nonetheless.

Contents

List of Figures

Chapter 1

INTRODUCTION TO THE PROBLEM: MOTIVATION AND GOALS

1.1. Introduction

A key issue for the field of Intelligent Tutoring Systems (ITSs) is how to treat students' bugs. When a student makes one, or possibly two bugs, the tutor only needs to decide what strategy to use with the student, e. g., point the bug out; fix the bug; hint at the existence of the bug. When a student makes several bugs, however, the tutor's problem is more complex. In addition to deciding what strategy to use, the tutor now must think about the order in which to tutor the bugs, which bugs to tutor together, which bugs to spend less time on in order to spend more time on other bugs, and so on. Each of these decisions can have an impact on how well the tutor helps the student.

The simplistic strategies that have been used to handle the problem of multiple bugs in existing ITS are only interim solutions. These strategies, which are discussed later in this chapter, were not intended to, and cannot be expected to, scale up to general solutions for the problem of multiple bugs in a wide variety of tutoring domains. For example, there are many patterns of multiple bugs that have implications for tutoring

strategies: few such patterns have been identified in existing approaches.

The goal of this dissertation is therefore to investigate one of the tutorial issues that is ubiquitous in complex domains and which has potentially important implications for the the design of all tutoring systems for complex domains. That issue is multiple bugs. This introductory chapter attempts to accomplish four main goals. First, the issue of multiple bugs is introduced and some intuitions are provided about why the study of multiple bugs is important for ITS. Second, five major decisions are described that confront the tutor who must help a student who has made multiple bugs. Third, the principle strategies are identified that have been used in an attempt to handle multiple bugs. Finally, the major claims and goals of the dissertation are identified.

1.2. The Three Problems of Multiple Bugs

When a student tries to solve a complex task, such as writing a computer program, it is unlikely that the first, second, or even last, attempt will produce a complete, bug free, solution. When novices try to solve a difficult problem they typically home in on a solution with a sequence of attempts in which more and more of the solution is correct and more and more of the bugs are identified and removed. Thus, a tutor who has the task of helping a student solve a complex problem, such as writing a 40 line

computer program, is almost always confronted with imperfect solutions that have multiple bugs. For example, empirical analyses of novice computer programmers show that students typically make six or seven bugs when they attempt to write a 40-line program that is assigned one-third of the way through an introductory programming course (Johnson, Cutler, Draper & Soloway, 1984; Spohrer et al. 1985).

Multiple programming bugs present three main problems for a tutor:

- First, the fact of multiple programming bugs requires the tutor to formulate a tutorial plan for how to help the student who made the bugs.

- Second, given exactly the same task specification, different students write different programs and the different programs contain different bugs. Thus, a tutor must be able to manage different students' solutions to the same problem where different solutions have different bugs. This type of variability is called inter-student variability.

- Third, the same bug can appear in conjunction with many different sets of other bugs. Because different contexts for the same bug can have different implications for the tutorial plan, the approach to a particular bug may depend upon the other bugs with which it appears. This type of variability is called contextual variability.

The main goal of this section is to describe the three chief aspects of the problem of multiple bugs.

1.2.1. Multiple Programming Bugs

The program in Figure 1.1 was written by a student in an introductory programming course and contains 9 bugs. This program is typical of those written by students trying to solve the Rainfall Assignment, shown in Figure 1.2. The tutor who wants to help the student who wrote this buggy program has many choices to make. For example, some bugs are more "important" than others because they cause the program to behave in ways that confuse the student --- BUG 4, the missing READ(RAINFALL); in the loop causes an infinite loop. An infinite loop makes the program look like it simply stopped in the course of execution and is very hard to understand. This is probably the reason the student came for help.

BUG 5, the misplaced update of TOTALRAIN enclosed in a BEGIN/END pair, suggests that the student has a misconception about the scope of a WHILE loop. BUG 5 is, therefore an "important" bug because tutors believe that misconceptions about programming concepts are important. Other bugs are "trivial" because they have virtually no effect on the behavior of the program and are fairly meaningless from the point of view of the student's understanding of programming. For example, BUG 9, the missing guard for an undefined Maximum which arises in the obscure case in which the user starts the program and terminates it before entering any data, is such a b Tutors would probably ignore his bug unless they wanted to help the student learn about

the importance of the skill of testing programs against boundary conditions such as the case of no valid input.

Further, the choice of strategy that the tutor uses to help the student with the bugs is not the same for each bug. For example, it may be enough to point out BUG 4 to the student and expect him or her to understand immediately that it is causing the computer to "hang" because of the infinite loop it creates. The misconception about how loops are scoped that is indicated by BUG 5, on the other hand, requires the tutor to do more than simply point it out. Thus, unless the tutor can handle all multiple bug situations with a few very simple rules, such as: "tutor all bugs"; "start with the first bug in the code"; "use the strategy of pointing out bugs" it is unlikely that existing approaches can be effective for complex cases of multiple bugs.

In fact, observations of experienced human tutors show that they do not use such a simplistic approach to tutoring multiple bugs (Littman, Pinto, & Soloway, 1985; Littman, Pinto, & Soloway, 1986; Littman & Soloway, 1986; Littman, Pinto, & Soloway, 1987). One very common plan that tutors employ to manage multiple bugs uses some of the student's simple bugs to build up to a more difficult bug (Littman 1988). For example, nearly all human tutors confronted with the program shown in Figure 1.1 reason that BUG 5, the misplaced update of TOTALRAIN, is the most serious bug and therefore should be the main topic of the tutoring session. However, rather than focusing the student's attention on that bug first, a tutor might look for another bug that would lead the student to think about the behavior of the

main loop, which seems to be the problem that led to BUG 5. BUG 3, the unconditional update of RAINDAYS, allows the tutor to lead the student to think about the behavior of the loop in preparation for tackling what appears to be the cause of BUG 5, namely a misconception about the scope of a WHILE loop. Before even working on the unconditional update, however, it might be advisable to eliminate the confusing infinite loop caused by BUG 4, the missing READ(RAINFALL); statement in the loop. A reasonable tutorial plan might therefore start with BUG 4, then move to BUG 3, and finally get to the key bug, BUG 5. The remaining bugs may or not be addressed, as time and the student's motivation permit.

If, on the other hand, the tutor used the simpler strategies of "start with the first bug in the code" and "tutor all bugs" and "point out the bugs" then the tutor could not drive the tutoring session toward the student's misconception about the scoping of loops (Littman, 1988). The student would simply see a sequence of bugs that needed to be fixed and would perhaps receive some simple advice about how to fix them. No attempt would be made to verify that the tutoring had clarified the student's misconception about the scoping of loops. Furthermore, the most trivial bugs, BUG 7, BUG 8, and BUG 9 would be addressed at the end of the tutoring session, perhaps making it hard for the student to remember all the lessons of the important bug, if the student even *realized* that it was the important bug. Thus, although this simplistic approach might be effective for bright students, or students who make trivial bugs for simple reasons,

it probably would not be effective for less able students or students who make bugs because they have misconceptions or are missing important knowledge.

In sum, a reasonable tutorial plan for a student's multi-bug program may not address all the bugs, may use different strategies for different bugs, and may tutor bugs in an order different from the order in which they appear in the student's program. In short, developing a plan for tutoring a multi-bug program is not always, or even often, straightforward.

```
PROGRAM NOAH (INPUT , OUTPUT);
CONST
SENTINEL = 99999;
VAR
RAINFALL, AVERAGE, RAINDAYS, TOTALRAIN : REAL;
BEGIN
(* PROMPT FOR  AND READ IN FIRST VALUE *)
WRITELN ('PLEASE TYPE IN THE FIRST VALUE');
READLN;
READ (RAINFALL);
(* INITIALIZE THE VARIABLE *)
BUG 1: Assignment of 0 to RAINFALL Clobbers Input of RAINFALL
RAINFALL := 0;  HIGHEST := 0;
BUG 2: Missing Initialization of RAINDAYS
(* READ IN NUMBERS UNTIL 99999 IS READ *)
(* ENTER DATA IN ONE DAY AT A TIME *)
WHILE RAINFALL <> 99999 DO
BEGIN
(* CHECK FOR HIGHEST RAINFALL *)
IF RAINFALL > HIGHEST THEN
HIGHEST := RAINFALL;
(* INCREMENT COUNT VARIABLES  DEPENDING ON RAINFALL VALUE *)
IF RAINFALL < 0 THEN
WRITELN ('ENTER ONLY POSITIVE NUMBERS')
ELSE
BUG 3: No Counter For Rainy Days -- RAINDAYS Increments Each Time
RAINDAYS :=  RAINDAYS + 1;
BUG 4: No READ(RAINFALL) In Loop
END;
BUG 5:  Update of TOTALRAIN Below Loop
BEGIN
TOTALRAIN := RAINFALL + TOTALRAIN;
END;
(* COMPUTE THE AVERAGE RAINFALL *)
BUG 6: Divide By Zero Guard Missing
BEGIN
AVERAGE := TOTALRAIN/RAINDAYS;
END;
(* PRINT OUT THE RESULTS *)
WRITELN;
WRITELN('THE PROGRAM READ IN ', RAINDAYS :0:2, 'RAINY DAYS');
BUG 7:  Wrong Variable Output
WRITELN('THERE WERE  ', RAINFALL :0:2, 'RAINY DAYS IN PERIOD');
BUG 8: No Guard for Undefined Average
WRITELN('THE AVERAGE WAS ',  AVERAGE :0:2, 'INCHES PER DAY');
BUG 9: No Guard for Undefined Maximum
WRITELN('THE MAXIMUM WAS  ',  HIGHEST :0:2, 'INCHES');
```

Figure 1.1: A Sample Buggy Rainfall Program

The Noah Problem: Noah needs to keep track of the rainfall in the New Haven area to determine when to launch his ark. Write a program so he can do this. Your program should read the rainfall for each day, stopping when Noah types "99999", which is not a data value, but a sentinel indicating the end of input. If the user types in a negative value the program should reject it, since negative rainfall is not possible. Your program should print out the number of valid days typed in, the number of rainy days, the average rainfall per day over the period, and the maximum amount of rainfall that fell on any one day.

Figure 1.2: The Rainfall Assignment

1.2.2. Inter-Student Variability -- Different Students: Different Programs

The statement that 200 students write 200 different programs is only somewhat exaggerated. For example, Johnson, et al. (1984) present a taxonomical catalogue of the most common bugs that students make when they solve the Rainfall Task. Each of the approximately 90 buggy programs presented in the catalogue was written by a student who was trying to solve the Rainfall Task: No two of the programs are the same.

Figure 1.3 shows fragments of two buggy solutions to the Rainfall Task. Each of the two multi-bug solutions has different bugs and each solution results from the student making different choices at the same point in the process of writing the program (Spohrer, 1989).

• In the first buggy solution the student has chosen to read an initial value of RAINFALL above the loop, then to process the value, and finally to get the next value of RAINFALL before checking the loop termination condition. This solution is called

the "Process-Read" solution because the input of the new RAINFALL occurs at the *bottom* of the loop.

- In the second solution the student's program reads the value of RAINFALL in the first statement of the loop, processes the value, gets another value, and so on. This solution is called the "Read-Process" solution because the input of the new RAINFALL occurs at the *top* of the loop.

These represent two different plans for reading and processing the value of RAINFALL.

Notice that the bugs in the two program fragments shown in Figure 1.3 are not the same. For example, in the first solution the student has added the incorrect value (RAINFALL) into the counter for TOTALDAYS. In the second solution the student has failed to guard the processing of RAINFALL against the case in which the value signals the end of processing. This bug cannot occur in the first solution because, as soon as a new value of RAINFALL is obtained, it is checked in the loop exit test. Thus, the two solutions have different bugs, in part because the students took different approaches to solving the same problem.

Just as different approaches to the same problem can lead to different bugs, the same approach to the same problem can lead to different bugs. For example, Figure 1.4 shows two more solutions to the Rainfall Task. Both students have used the Process-Read plan for the loop. Notice that in the first solution there is a spurious initialization of RAINFALL above the loop which is not present in the second solution. Equally, the second solution has a misplaced update of TOTALRAIN which is correctly

placed in the first solution. Thus, even though two students tried to solve the same task "the same way", they made different bugs.

A tutor must be able to manage the inter-student variability which is the rule rather than the exception.

```
PROGRAM NOAH-1 (INPUT , OUTPUT);
        . . .
READ (RAINFALL);
WHILE RAINFALL <> 99999 DO
BEGIN
IF RAINFALL > HIGHEST THEN
HIGHEST := RAINFALL;
BUG 1: RAINFALL Added to TOTALDAYS Counter
TOTALDAYS := TOTALDAYS + RAINFALL;
READ (RAINFALL);
BUG 2: No Update of TOTALRAIN
END;
        . . .


PROGRAM NOAH-2 (INPUT , OUTPUT);
        . . .
RAINFALL ;= 0;
WHILE RAINFALL <> 99999 DO
BEGIN
READ (RAINFALL);
BUG 1: No Guard for RAINFALL = Sentinel
IF RAINFALL > HIGHEST THEN
BUG 2: Assignment Backwards
RAINFALL := HIGHEST;
TOTALRAIN := TOTALRAIN + RAINFALL;
END;
        . . .
```

Figure 1.3: Two Solutions to the Rainfall Task

```
PROGRAM NOAH-1 (INPUT , OUTPUT);
        ...
BUG 1: Missing Initialization of TOTALRAIN
READ (RAINFALL);
WHILE RAINFALL <> 99999 DO
BEGIN
IF RAINFALL > HIGHEST THEN
HIGHEST := RAINFALL;
READ (RAINFALL);
END;
BUG 2: Misplaced Update of TOTALRAIN
BEGIN
TOTALRAIN := TOTALRAIN + RAINFALL;
END;
        ...


PROGRAM NOAH-2 (INPUT , OUTPUT);
        ...
READ (RAINFALL);
BUG 1: Assignment of 0 to RAINFALL Clobbers Input of RAINFALL
RAINFALL := 0;
WHILE RAINFALL <> 99999 DO
BEGIN
IF RAINFALL > HIGHEST THEN
HIGHEST := RAINFALL;
BUG 1: TOTALRAIN Wrongly Increased 1 Instead of TOTALRAIN
TOTALRAIN := TOTALRAIN + 1;
READ (RAINFALL);
END;
        ...
```

Figure 1.4 : Two Solutions to the Rainfall Task

1.2.3. Contextual Variability -- Same Bug: Different Context

The approach of a tutor to a particular bug can depend on the other bugs that appear with the bug. For example, Figure 1.5 shows the same bug associated with different groups of other bugs. In both cases the student failed to initialize the variable

MAXRAIN (BUG 1) which keeps track of the maximum value of RAINFALL. However, a tutor would approach the same bug differently according to which of the two programs in which it appeared:

- In the first program, the only problem with the student's attempt to determine, and output, the maximum value for RAINFALL is the missing initialization of MAXRAIN. Indeed, several other variables are also incorrectly left uninitialized. In this case, the tutor would probably deal with all of the missing initializations at the same time without singling out the missing initialization of MAXRAIN.

- In the second program there are three bugs associated with the student's attempt to calculate MAXRAIN. First, MAXRAIN is not initialized: This is the same bug that appeared in the first program. Second, the boolean test for the maximum is incorrect: The test should be "<>" rather than "<". Third, the statement to write out the value of MAXRAIN is missing from the program.

In this case the tutor would probably put all the bugs associated with the calculation of MAXIMUM into the same "group". That is, the tutor would not address the missing initialization of MAXRAIN along with the other missing initializations, as in the first program shown in Figure 1.5. Instead, the tutor would address the missing initialization of MAXRAIN along with other bugs in the student's attempt to calculate and output the value of MAXRAIN.

In sum, the context in which a bug appears can affect how a tutor approaches it. Thus, the treatment of a particular bug in

the student's program can depend upon the treatment of the other bugs in the program.

```
PROGRAM NOAH-1 (INPUT , OUTPUT);
        ...
BUG 1: No Initialization of MAXRAIN
BUG 2: No Initialization of TOTALRAIN
BUG 3: No Initialization of TOTALDAYS
        ...
READ (RAINFALL);
WHILE RAINFALL <> 99999 DO
BEGIN
IF RAINFALL > MAXRAIN THEN
MAXRAIN := RAINFALL;
BUG 4: RAINFALL Added to TOTALDAYS Counter
TOTALDAYS := TOTALDAYS + 1;
READ (RAINFALL);
END;
        ...
WRITELN ('Maximum Rain Was ', MAXRAIN);


PROGRAM NOAH-2 (INPUT , OUTPUT);
BUG 1: No Initialization of MAXRAIN
BUG 2: No Initialization of TOTALRAIN
BUG 3: No Initialization of TOTALDAYS
        ...
READ (RAINFALL);
WHILE RAINFALL <> 99999 DO
BEGIN
BUG 4: Wrong Boolean Test for MAXRAIN
IF RAINFALL < MAXRAIN THEN
MAXRAIN := RAINFALL;
TOTALDAYS := TOTALDAYS + 1;
TOTALRAIN := TOTALRAIN + RAINFALL;
READ (RAINFALL);
END;
BUG 5: No Output of MAXRAIN
        ...
```

Figure 1.5: The Same Bug in Different Contexts

1.3. The Tutor's Task

As the foregoing examples show, when a student makes several bugs the tutor must reason about how to construct a tutorial plan that coordinates the tutoring of the bugs: Some bugs should be treated together, some bugs should be addressed before others, some bugs should go untutored, and so forth. In effect, the tutor must construct a tutorial plan to handle multiple bugs (Littman, 1988).

The tutorial planning task requires the tutor to address, at least, the five Tutorial Planning Decisions shown in Figure 1.6:

• Tutorial Planning Decision 1: **GROUPING** *Relationships among the bugs* -- Tutors do not treat all bugs as distinct. They classify bugs according to several criteria including e. g., what caused the bugs, the effects they have on the program, what the student needs to learn to avoid making the bug in the future, and so forth. This classification of bugs provides a basis for deciding which bugs to group together in the tutorial plan.

• Tutorial Planning Decision 2: **YES-NO** *Whether to tutor each bug* -- Some bugs are critical, others are too trivial, or too hard, to tutor; other bugs offer diagnostic opportunities to determine the student's skill level and still others provide the starting point for helping the student to extend his or her knowledge about e. g., debugging methods.

• Tutorial Planning Decision 3: **ORDERING** *When to tutor each bug* -- Bugs seem to fit naturally in the tutoring plan at

some points but not at others. For example, a student who is unmotivated, or lacks confidence, would probably benefit from starting with a simple bug that is easy to fix. With a highly motivated, confident student, on the other hand, the tutor could probably start with a difficult, important bug without running the risk of discouraging the student.

• Tutorial Planning Decision 4: **OBJECTIVE** *What to accomplish* Sometimes tutors just want to fix a bug quickly to get it out of the way; other times tutors want to focus on a bug to clear up a misconception. In generating a tutorial plan for a student's bugs, tutors reason about, for example, what they want the student to get out of the tutoring session.

• Tutorial Planning Decision 5: **STRATEGY** *How to tutor each bug* -- Some bugs can simply be pointed out to the student; others are best approached with a more complex strategy e. g., asking the student to generate a justification for why the buggy code was written as it was and then asking the student to generate an alternative that avoids the bugs. As well, some students need considerable help fixing bugs, others just a hint about the right direction for a solution.

These five Tutorial Planning Decisions form the backbone of the tutorial planning process: Making the five Tutorial Planning Decisions results in a tutorial plan. A tutorial plan for multiple bugs consists of 1) bugs grouped together to facilitate tutoring; 2) bugs prioritized so that, for example, trivial bugs will be overlooked if more important bugs take up most of the tutoring session, 3) bugs ordered so that, for example, tutoring bugs early

in the session will lay the groundwork for tutoring other bugs later in the session, 4) objectives established for the treatment of each of the bugs, or groups of bugs, and 5) tutorial strategies appropriate for each of the bugs. The next section examines three possible approaches to the five Tutorial Planning Decisions. The examination shows that they are inadequate, especially if the research goal is to devise solutions that are generally applicable to ITS for many domains.

Tutorial Planning Decision	Label	Description
1	GROUPING	Which bugs to tutor together?
2	YES-NO	Should the bug be tutored?
3	ORDERING	What order to tutor groups?
4	OBJECTIVE	What to achieve by tutoring?
5	STRATEGY	What strategies to use?

Figure 1.6: The Five Tutorial Planning Decisions

1.4. Inadequacy of the Brute Force Approach

Given that the tutor must generate a plan to help students who make multiple bugs, the question is how such plans are generated. It turns out that, if a student makes the 9 bugs shown in Figure 1.1 and the tutor must decide only a) the order in which to tutor the bugs (Tutorial Planning Decision 2) and b) which of four possible teaching strategies to use (Tutorial Planning Decision

5), that there are several billion possible alternative arrangements of orders and strategies i. e., plans.

Because the size of the search space of tutorial plans is so large it is unreasonable to suppose that tutors use a brute force approach to tutorial plan generation. Even if the human, or machine tutor could generate all, or many of, the possible tutorial plans there remains the problem of deciding which among them to use. For example, what heuristics would the tutor use to select one plan in preference to all the others? What knowledge would allow the tutor to reject certain plans as "obviously" inappropriate?

Because of these considerations, it does not seem reasonable to pursue the brute force approach to tutorial planning for multiple bugs in machine tutors. Rather, an effective approach to the problem of tutorial planning for multiple bugs seems to require a knowledge-based approach. Indeed, the evidence presented in this dissertation suggests that, if a tutor has the right knowledge, the search process plays virtually no role in tutorial plan generation.

1.5. Previous Work on Multiple Bugs

Previous approaches to the problem of multiple bugs can be classified into two main categories. First, some ITS simply avoid the problem of multiple bugs by preventing students from making more than one bug at a time. Second, other ITS capitalize on the

features of the knowledge being taught so that a few simple rules for multiple bugs are adequate for the domains in which they tutor.

1.5.1. Preventing Multiple Bugs

One approach to the problem of multiple bugs is to prevent the student from making them. For example, the intelligent tutoring systems built by John Anderson and his colleagues (Anderson, Boyle, Farrell, and Reiser, 1984) adhere strongly to the principle that students should receive immediate feedback whenever they make a mistake and should not be allowed to proceed until the mistake is corrected.

There are some advantages to the tutor and to the student of the immediate feedback approach. For the tutor the prevention of multiple bugs eliminates the problem of figuring out how to handle them. The problem of devising a tutorial plan for the bugs does not arise. The tutor does not have to reason about, for example, 'similar' bugs, or 'interacting' bugs. Instead, once a bug is detected, the tutor only needs to decide what to do or say.

For the student, preventing multiple bugs can have the advantageous result that the student's finished product is free of bugs. Thus, in the case of the LISP tutor, when the student finishes a program it is guaranteed to run.

There are, however, some potential disadvantages to the prevention of multiple bugs.

- First, it may actually be good for students to make more than one mistake before receiving corrective feedback. Observations of human tutors, and intuition, suggest that students benefit from making, finding, and correcting their mistakes and that a tutor is most constructive if it intervenes when the student is unable to find and correct important errors rather than nattering the student at every mistake.

- Second, it is not always advisable to interrupt each time the student makes a bug. For example, if a student is deep in the analysis of a complex problem and makes a typographical error, interrupting could cause the student to lose an important train of thought. In addition, as the WEST experience demonstrated, interrupting after each mistake can cause students to feel badgered and therefore to lose interest (Burton and Brown, 1982).

- Third, it seems unlikely that it will be possible to provide the kind of exhaustive analysis of the problem solving processes for complex domains that the immediate feedback strategy requires. For example, the use of an immediate feedback strategy in Anderson's LISP programming tutor depends on a complete, unambiguous analysis of all the solution paths for each program that a student could write. This strategy has been possible for the LISP tutor because of the small size of the LISP programs that the tutor can help with. The programs for which the LISP tutor can provide coaching is an order of magnitude smaller than the programs students write midway through an introductory PASCAL programming course. If larger programs lead to ambiguity in the tutor's diagnosis of the cause of a

student's bug then it is impossible to provide immediate, accurate, intervention. If the tutor waits for more information to disambiguate the diagnosis, then it is possible for the student to make more bugs before the tutor receives sufficient disambiguating information. This would require the tutor to decide what to do about the additional bugs: tutor them first? tutor them last? ignore them? pick the most important bug so far and tutor it? pick an easy bug if the student is not very skilled? tutor the bugs in the order in which they were created? in the order in which they are encountered when the program executes? according to the underlying cause?

In sum, though the strategy of preventing students from making multiple bugs has some definite advantages, the immediate feedback and prevention strategy does not appear to be one that will scale up effectively to more complex problem solving situations.

1.5.2. Domain Independent Rules For Tutoring Multiple Bugs

A second approach to multiple bugs uses a few general, simple rules that depend on intrinsic structure of the knowledge students are trying to acquire. For example, in the case of the Socratic tutors developed by Collins and his colleagues, the structure of the domain imposes a corresponding structure on tutoring. In the weather domain, for example, Collins, Stevens, and Goldin (1982) have identified several "scripts" that characterize the causal structure, and therefore the knowledge

that a tutor should teach, of complex weather processes. For example, the "heavy rainfall" script is represented as a sequence of events in which 1) a large warm air mass absorbs evaporated water from a large water mass, 2) winds carry the warm air mass over land, 3) the warm, water-laden air mass cools quickly over land, and 4) the cooled moisture in the air mass precipitates onto the land area. Tutoring students who have buggy knowledge about e.g., the role of evaporation of ocean water on weather in regions of heavy rainfall is directed toward teaching, or clarifying, relevant components of, and relations among components of, such scripts (Collins, et al., 1982).

In the weather domain much of the buggy knowledge that students have can be viewed in terms of failures to understand causal scripts. For example, a student might think that a water-laden cloud would be too heavy for a wind to move it past a coastline and over land. The use of such scripts to teach students capitalizes on the structure of the scripts and thus leads naturally to rules for managing multiple bugs. For example, there is no point in teaching the student about a *new* component of the script if the student has an erroneous belief about a component on which it depends; equally, there is no value in teaching a small detail of a causal script if the student has a misunderstanding about the large issue in which the detail is based. Thus, 'teach knowledge preconditions' is a simple rule for tutoring that applies when causal scripts organize the domain knowledge that a student must master.

Concretely, if the tutor is trying to teach the student why it is so rainy in Oregon, a warm coastal region, and if the student does not know that evaporation of warm ocean water is a crucial, causal determinant of weather patterns in coastal areas, there is not much point in teaching the student that the temperature of ocean currents indirectly affects evaporation rates. Thus, because of the intrinsic structure that the domain provides --- the causal scripts for weather processes --- simple rules often suffice for determining when to tutor bugs.

In sum, there are tutoring situations in which the impact of multiple bugs can be minimized. Avoiding the problem of multiple bugs, however, requires either that 1) the behavior of the student is sufficiently unambiguous to permit the tutor to make correct interventions with immediate feedback or 2) the structure of the knowledge the student is trying to acquire imposes natural restrictions on how to help the student with multiple bugs. It therefore appears that a more general approach is needed to the problem of multiple bugs because many tutoring domains, such as computer programming, do not conform to the foregoing requirements.

1.6. Main Claim and Goals of Dissertation

The purpose of the dissertation is to provide support for the thesis that tutorial planning for multiple bugs is not haphazard

but, rather, is governed by well-defined knowledge about programming and tutoring.

The main claim of the dissertation is therefore:

MAIN CLAIM: To effectively tutor a student who makes multiple bugs,

- a tutor must generate a tutorial plan for the tutoring session
- the tutorial plan must represent, at least, whether to tutor each bug, when to tutor each bug, and how to tutor each bug
- the tutorial plan is generated by making the five Tutorial Planning Decisions.

To support this claim the dissertation has three main goals, ranked in order of the emphasis they have been given in the research for the dissertation:

GOAL 1: Identify Knowledge Required for Tutorial Planning

Generating a tutorial plan for multiple bugs requires extensive knowledge about e.g., causes of bugs, bug importance, tutorial strategies, and so on. The problem of tutoring multiple bugs has not been systematically studied and current solutions are simplistic and hence restricted in generality. A primary goal of the dissertation is, therefore, to identify the knowledge that tutors use to plan for multiple bug tutoring in the programming domain.

GOAL 2: Characterize The Generation of Tutorial Plans

Once the knowledge has been identified that is required to generate tutorial plans, it remains to determine how the knowledge is coordinated to generate plans. That is, what control structure governs the use of the knowledge? The second goal of the dissertation is to characterize the generation of tutorial plans. This resulted in a computer program which generates the same tutorial plans for multiple bugs as human tutors. The tutorial planner generates 1) the same plans for tutoring novices' multiple bug programs that experienced human tutors generated in an empirical study of human tutors and 2) plausible tutorial plans for many additional multiple bug novice programs. As discussed in a later section, the plans for several of the latter programs have been validated in an empirical study of plan acceptability.

GOAL 3: Investigate Recovery from Tutorial Failures

Once the tutor has constructed a tutorial plan, the plan must be carried out. The execution of a tutorial plan sometimes leads to failures and the need to recover from the failures. Though not a major goal of the dissertation, some preliminary analyses were made of types of tutorial failures and strategies for recovering from failures. The analyses were based on the behavior of human tutors. Some of the results of this work were incorporated into the plan execution mechanism of the tutor. The main goal of the work on tutorial failure is to lay the groundwork for future

research on the delivery of tutoring that follows tutorial planning.

The research on strategies for multiple bugs was carried out in the domain of novice PASCAL programming. The main focus of the dissertation is thus how to handle multiple bugs in the programming domain. Many of the strategies for handling multiple programming bugs are applicable to other domains. Discussion of the domain generality of this research is provided at several points in the dissertation.

1.7. The Dissertation in a Nutshell

"Strategies for Tutoring Multiple Bugs" addresses five primary topics.

- The five Tutorial Planning Decisions
- The knowledge required to make the five Tutorial Planning Decisions
- The computer program, TP, that generates tutorial plans like the tutorial plans that experienced human tutors generate
- The empirical study that was designed to provide evidence for the validity of the model of tutorial planning
- Recovering from failure during tutorial plan execution.

These five topics form the skeleton of the dissertation; this section provides a brief description of the work on the five topics:

First, five decisions that must be made to create a tutorial plan for multiple bugs are identified and described. These decisions are the five Tutorial Planning Decisions that were discussed earlier in this chapter. The result of making the five Tutorial Planning Decisions is a tutorial plan that reflects each of the five Tutorial Planning Decisions. That is, a tutorial plan specifies which bugs to address, which bugs to address together, when to address each bug, the goals of addressing each bug, and the strategy to use to achieve the goals.

Second, the knowledge required to make the five Tutorial Planning Decisions is described in detail and shown to fall into four main categories: 1) knowledge about students; 2) knowledge about the domain of programming; 3) knowledge about bugs; 4) knowledge about teaching. Each of the four knowledge categories has several types of knowledge within it. For example, within Knowledge Category 1 is knowledge about student ability; knowledge about the causes of bugs is in Knowledge Category 3.

Third, TP, the computer program that creates human-like tutorial plans is described. TP uses the knowledge in the four categories to construct tutorial plans. The control structure of TP is a planner that has five goals. Each of the five goals corresponds to one of the Tutorial Planning Decisions. Creating a tutorial plan begins with a classification task in which knowledge within several of the above-noted four categories is used to decide which bugs to place in the same group. There are approximately 12 types of groups. Each type of group is defined by the relationships among the two or more bugs in the group. For

example, if a student leaves out all the initializations in a program then, ceteris paribus, all the bugs would be placed in the same group because they are all the same type of bug (missing) and they are all in the same part of the program (initialization). The 12 different types of groups play a central role in the generation of tutorial plans because the remaining four Tutorial Planning Decisions -- whether to tutor bugs, when to tutor them, why to tutor them and how to tutor them are all affected by the groups into which the bugs are placed.

Fourth, the empirical validation of the tutorial planner is discussed. The empirical validation focused on whether experienced human tutors found the tutorial plans for multiple bugs generated by the computer program, TP, to be acceptable. Experienced tutors answered questions about tutorial plans generated by the computer program for 10 multi-bug PASCAL programs. Judgements of acceptability were made on several features of the tutorial plans e. g., whether the tutorial plans were as good as the plans the human tutors would have generated; whether the human tutors would be willing to use the tutorial plans to help the student who wrote the program; and so forth. The results of the empirical validation support the conclusion that the plans generated by the tutorial planner are "the same as" those generated by experienced human tutors.

Fifth, the problem of recovering from failure during tutorial plan execution is addressed. The analysis of failure recovery suggests that there are two main types of failures, namely major failures and minor failures. Each specific failure has an

associated recovery strategy which is used when the failure occurs. For example, if a minor failure arises in which the student cannot locate a bug when the tutor describes the bug type -- say the tutor tells the student that there is a missing guard on an update -- then the tutor might recover from the failure by telling the student where the bug is in the program text e. g., the recovery strategy is to identify for the student the line on which the bug occurs.

The chapters that focus on each topic are identified in the next section. Each topic is treated in one or more chapters in the dissertation. The dissertation also considers several additional, subsidiary topics, as the next section describes.

1.8. Contents of Dissertation

"Strategies for Tutoring Multiple Bugs" contains nine chapters following the introduction:

• Chapter 2 and Chapter 3 present the results of the analyses of the empirical studies of human tutorial planning for multiple bugs in the programming domain. The main content of the chapters is a description of the knowledge types in the four knowledge categories identified in the previous section. The primary focus of this section is therefore GOAL 1, above.

• Chapter 4 discusses different tutorial plans for the same bugs and the tutorial planners that generate the different plans. The discussion in this chapter focuses on a sequence of

increasingly complex tutorial plans for the same set of bugs and shows 1) the types of knowledge required to make the tutorial planning decisions that result from the five Tutorial Planning Decisions and 2) the different control structures that are required to make use of the knowledge to generate the different plans.

• Chapter 5 describes TP, the tutorial planner that is the demonstration program for the thesis. This chapter describes 1) the representations for TP's knowledge and 2) the control structure which governs the generation of TP's tutorial plans. Chapter 5 therefore focuses on **GOAL 2**, identified above.

• Chapter 6 describes the strength of the current implementation of the tutorial planner. This chapter discusses the programming assignments for which TP can generate tutoring plans and the 140 or so bugs about which TP can reason.

• Chapter 7 touches on issues that arise during plan execution. Although plan execution was not intended to be a major focus of the dissertation, this chapter discusses some observations about how human tutors recover from tutorial planning failures and how these observations were incorporated into the tutorial planner. Chapter 7 thus addresses **GOAL 3**, above.

• Chapter 8 briefly addresses the generality of the current approach to multiple bugs. Although the tutorial planner depends on knowledge about the domain of computer programming, much of its reasoning is independent of any particular domain. This

section identifies some of the domain general knowledge of the tutorial planner as well as some of its limitations.

- Chapter 9 presents the main conclusions that can be drawn from the work presented in the dissertation.

- Chapter 10 identifies several directions for future research that have emerged during the work on tutorial planning for multiple bugs. The future directions range from obvious next steps, such as studying a wider sample of tutors and programming assignments, to more ambitious topics of research, such as generalizing the tutor to other domains and integrating it with a complete tutoring system for novice programming.

The most important chapters for the non-technical reader are Chapters 2, 3, 4, 5, and 9. Readers interested in the preliminary work on tutorial plan execution may wish to read Chapter 7.

Chapter 2

DESCRIPTION OF TUTORIAL KNOWLEDGE

2.1. Introduction

Chapter 1 introduced the problem of multiple bugs in tutoring and provided some intuitions about how tutors, whether human or machine, could tutor effectively in multi-bug situations. The main suggestion of that discussion was that tutors engage in knowledge-intensive planning to determine how to help a student who has made multiple bugs. For example, it was suggested that, in generating tutorial plans for multiple bugs, tutors reason about the student's ability, the importance of bugs, the effects of bugs on the program, why students make bugs, relationships among bugs, and so on.

The primary purpose of this chapter and the next chapter is to describe the knowledge that tutors use to generate tutorial plans for multiple bug tutoring situations. These two chapters therefore have three main goals. The first goal is to show that the knowledge that tutors use to generate tutorial plans is not a haphazard collection but, rather, can be organized into four main categories. For example, knowledge about the student is one such category of knowledge. The second goal is to present the details of each of the types of knowledge. For example, one type of tutorial knowledge that tutors use to generate tutorial plans is about the causes of bugs; this

chapter describes the causes of bugs in detail. In essence, the goal of the presentation of the details of the tutorial knowledge types is to display the content theory of tutorial planning for multiple bugs. The goal of the next chapter is to show how the five Tutorial Planning Decisions depend on the tutorial knowledge types described in this chapter.

The remainder of this chapter is divided into three main sections. First, a brief description is given of the empirical basis for the work that is reported in the dissertation. The second section provides an overview of the four knowledge categories, the tutorial knowledge types in them, and how the five Tutorial Planning Decisions are based on the types of knowledge. The third section describes in detail the three tutorial knowledge types on which tutorial planning depends.

The following chapter describes how the five Tutorial Planning Decisions depend on the tutorial knowledge types described in this chapter. The section that describes the tutorial knowledge types required for tutorial planning precedes the section on the five Tutorial Planning Decisions because the rules that make the five Tutorial Planning Decisions depend upon the tutorial knowledge types.

2.2. Empirical Basis of the Research

The description of the types of knowledge that tutors use to generate tutorial plans is based on empirical studies of human

tutors (cf. Littman, Pinto, & Soloway, 1987). The methods used to collect data from tutors placed experienced tutors in situations that were similar to real tutoring situations. Three studies were performed and all the results described in this chapter derive from one or all of the studies:

- In the first study, called the Real Time Study, tutors were observed and videotaped in real time interactions with students. The purpose of the real time tutoring studies was to generate information about how tutors behaved when students came to them with multi-bug programs.

- In the second study, called the Questionnaire Study, experienced tutors were placed in an experimental situation designed to elicit the reasoning they perform when they must decide how to tutor a student who has made multiple bugs (Littman, Pinto, & Soloway, 1987). In the experiment tutors were shown five buggy solutions to the Rainfall Assignment that had been written by students in an introductory programming class. Each program had between six and nine bugs. The bugs were all identified for the tutors. For each program, the tutor was asked to fill out a questionnaire that asked about 1) the tutorial plans they developed to treat the bugs and 2) the reasoning behind the plans. The questionnaire thus asked them to say what bugs they would tutor together, and why; what bugs were most important, and why; why they thought the student made the bugs; what bugs they would tutor first, second, etc. -- and why; how skilled they thought the student was who wrote the program; and so forth. A portion of the questionnaire is shown in Figure 2.1.

- In the third study, called the Failure Recovery Study, tutors were observed while they tutored students who were enrolled in an introductory PASCAL programming class. Each time the student failed to understand the point that the tutor was making the situation and the failure were described, and the corrective action taken by the tutor was characterized. This study provided the basis for the preliminary work on tutorial plan execution and recovery from failure that is described in this dissertation.

One issue will continue to appear throughout the discussion of the knowledge categories and the five Tutorial Planning Decisions in this and the next chapter. That issue is that the descriptions of how tutors resolve the five Tutorial Planning Decisions often sound as though every tutor makes each decision in the same way. This is, of course, not so. Littman, Pinto, & Soloway (1988; 1989) show that, statistically, different tutors' resolutions of the five Tutorial Planning Decisions are very far from random. There are, of course, differences among tutors. Littman, Pinto, & Soloway (1985), for example, discuss in detail some differences in tutors' grouping and ordering of bugs in a single buggy program. The intent of the dissertation, however, is not to present an account of individual differences among tutors in making the five Tutorial Planning Decisions: That is a topic for future research. Rather, the intent of the research reported in the dissertation is to account for the predominant patterns of decisions in resolving the five Tutorial Planning Decisions, as measured by the methods reported in Littman, Pinto, & Soloway (1988; 1989) as guides.

We would like to get a picture of your approach to tutoring. Please answer each of the following questions. These questions are intended to prompt you rather than constrain you; feel free to include other remarks as well.

a) Which bug(s) would you tutor first?

If applicable, why did you group these bugs together?

b) Why would you tutor this bug (or group of bugs) first?

c) Why do you think the student might have made this bug (or group of bugs)?

d) Is there anything you could find out from the student about whether or she knows in relation to this bug (or group of bugs) that would make you more certain of what the misunderstanding is? What? Why?

e) How would you approach tutoring a student who made this bug (or group of bugs) only?

f) What would you say are the main points that you want the student to learn from your tutoring this bug (or group of bugs) ?

g) If you had to tell the student why this is a bug (these are bugs), what would you say? Would you do anything before you told the student? If you would do something, what would you do?

h) Given what you've accomplished with the student so far, would you tell the student to go re-write the program independently or would you continue tutoring the student? Why?

i) If you would tell the student to re-write the program independently, what would you expect to happen? Would you expect to tutor the bugs which might still remain unaddressed or would you expect not to address those bugs at all? Explain.

Figure 2.1: Portion of Tutors' Questionnaire

2.3. Synopsis of Knowledge Categories and Tutorial Planning Decisions

The purpose of this section is to provide an overview of the categorization scheme for the types of knowledge required to make the five Tutorial Planning Decisions identified in the first chapter.

For example, when a tutor decides that the two missing output guard bugs shown in Figure 1.1 should be grouped together for tutoring (Tutorial Planning Decision 1), the tutor reasons about a) relationships among bugs -- each bug is a missing plan component -- and b) types of plan components -- the plan component is the guard against the case of no input. During the course of introducing the problem of tutoring multiple bugs, the discussion and examples of Chapter 1 mentioned approximately a dozen types of knowledge that tutors use to make the five Tutorial Planning Decisions.

Figure 2.2 shows the tutorial knowledge types mentioned in Chapter 1 along with example quotations that illustrate the knowledge types. For example, the first row in the figure gives an example of a tutor stating that a misconception caused the bug under consideration. Tutors use knowledge about Bug Cause to make several Tutorial Planning Decisions: A tutor would treat a bug caused by a slip differently from a bug caused by a deep misconception. The example for tutorial strategy shows that the tutor would use hand simulation of code to show the student how an action performed at line 23 has the effect of negating an action performed at line 16 before the action taken at line 16 has its proper effect. Although it is useful to see a list of the knowledge types and an example illustrating a tutor reasoning about each, it is hard to form from this listing a coherent picture of the knowledge required for tutorial planning. The knowledge must be categorized.

2.3.1. Categorizing the Knowledge Types

As it happens, each of the knowledge types shown in Figure 2.2 can be placed naturally in one of four categories. Figure 2.3 identifies the four categories of knowledge and the types of knowledge in each of the four categories. The columns in the body of the table contain the types of knowledge that fall into each category. Thus, *ability* is a type of knowledge that tutors have about *students* and this knowledge type is therefore placed in that category. As will be shown later in this chapter, the knowledge types shown in the figure are the basis of the five Tutorial Planning Decisions and each of the knowledge types typically affects more than one of the decisions. The four knowledge categories that organize the tutorial knowledge types are:

- Knowledge Category 1: Knowledge about Students For example, tutors know about different levels of student ability and this has implications for selecting a tutorial strategy: A student who is very able may be tutored with a strategy that requires more work from the student than a strategy that would be used with a student who is less able.

- Knowledge Category 2: Knowledge about the Domain For example, in the domain of programming, tutors know the programming plans that can be used to solve the tasks that students receive as assignments and they frequently focus on teaching students the plans that are better than the ones that the student used to try to solve a problem. As an example, tutors many know different plans for creating a main processing loop for e.g., the Rainfall Task and they are concerned to teach students the conditions in which they should use the different plans.

• Knowledge Category 3: Knowledge about Bugs Tutors know about patterns of bugs and relationships among bugs. For example, tutors are sensitive to the pattern in which the student unknowingly creates a second bug in the attempt to repair an original bug that the student detected. As will be seen later, there are approximately 15 types of patterns of bugs that form a hierarchy. This knowledge is used, in part, to determine bug groupings, Tutorial Planning Decision 1.

• Knowledge Category 4: Knowledge about Teaching Tutors have two main types of knowledge about teaching. First, tutors know how to construct a tutorial plan: This is the main topic of the dissertation. Second, tutors know how to execute a tutorial plan and, especially, how to recover from failures that occur during tutoring. The topic of recovery from failure while executing tutorial plans is addressed later in the dissertation. Tutorial strategies is one of the main knowledge types that tutors use to construct tutorial plans. Tutors know when particular tutorial strategies, such as asking the student for an explanation of a choice of plan, are appropriate. Knowledge about different tutorial strategies, and when they are appropriate, therefore affects the fifth Tutorial Planning Decision.

Some of the knowledge types tutors use to generate tutorial plans are simple. For example, student ability is either low, moderate, or high. Other knowledge types are complex. For example, one of the complex types of knowledge that tutors have about the domain of programming is task specifications (symbolized as "task spec" in the table): Task specifications for a single assignment,

such as the Rainfall Assignment, typically specify more than 30 goals that are related to one another in several possible ways e. g., some goals are preconditions of other goals; some goals are subgoals of other goals, and so forth.

All the knowledge types are described later in this chapter.

KNOWLEDGE TYPE	EXAMPLE QUOTATION
Cause of Bug	"Badly formed internal concept of conditionals."
Tutorial Strategy	"Hand Simulation to show line 23 clobber 16."
Effect of Bug	"Line 23 clobbers line 16 before it should."
Student Ability	"... understands how to program pretty well."
Importance of Bug	"These are the most serious bugs."
Patterns of Bugs	"These are all basically the same bug."
Programming Plans	"I want to show them the right way to do ..."
Student Knowledge	"He doesn't know the REPEAT loop."
Teaching Goals	"... how to construct a WHILE-DO loop."
Program Concepts	"If then else is sometimes hard ..."
Task Specs	"... calculation of the average is really the focus"
Teaching Rules	"I like to get to the hard bugs as soon as I can."
Student Motivation	"If student was feeling hopeless I would ..."

Figure 2.2: The Knowledge Types For Tutorial Planning

CATEGORY 1	CATEGORY 2	CATEGORY 3	CATEGORY 4
STUDENTS	DOMAIN	BUGS	TEACHING
ability	task specifications	causes	tutorial planning knowledge
motivation	programing plans	effects	plan execution knowledge
intentions	concepts	types	
knowledge	skills	plan location	

Figure 2.3: Tutorial Knowledge Types in 4 Categories of Tutorial Knowledge

2.3.2. Knowledge Types and Tutorial Planning Decisions

All the knowledge types shown in Figure 2.3 play a role in the generation of tutorial plans for multiple bugs. Each of the knowledge types is used to make at least one of the five Tutorial Planning Decisions. For example, student ability can affect the tutorial plan. If a student of high ability makes several similar bugs that are hard to find, the tutor might help the student find only one and then tell the student that there may be some additional, similar bugs. If the student is of medium ability the tutor may decide to help the student find all the bugs but only assist with the repair of one of the bugs. If the student's ability is low, the tutor may help the student find and repair all the bugs. Hence, the student's ability has an effect on Tutorial Planning Decision 5, the selection of a strategy for tutoring the student.

Figure 2.4 illustrates the dependence of the five Tutorial Planning Decisions on the types of knowledge in the four categories. The table presented in Figure 2.4 is intended just to give an idea of how some of the knowledge types contribute to the five Tutorial Planning Decisions. The treatment in this overview section is not intended to be exhaustive. The relationships between knowledge types and Tutorial Planning Decisions is elaborated in the last section of this chapter which describes the rules that make the Tutorial Planning Decisions and the knowledge on which the rules depend.

As an example of the dependence of Tutorial Planning Decisions on the knowledge types shown in Figure 2.3, consider a tutor's decision *not* to tutor BUG 8 and BUG 9 in the program shown in Figure 1.1. Figure 2.4 shows each of the five Tutorial Planning Decisions and the knowledge types that they depend on. The columns represent the four categories of knowledge. Each entry in a cell in the figure indicates that the Tutorial Planning Decision corresponding to that row uses knowledge of the type that is in the cell. As the figure suggests, Tutorial Planning Decision 2, whether to tutor the bugs, depends upon knowledge about the student, knowledge about the domain, and knowledge about bugs. A tutor's reasoning leading to the decision not to tutor BUG 8 and BUG 9 might go something like this:

• **Knowledge about the Student**: The student's ability is high, so the tutor does not have to worry that the student would be unable to fix the bugs later, without help, if they caused a problem.

• **Knowledge about the Domain**: The task specification goal of guarding the output against the rare case of no input is neither explicit in the problem description nor an important goal for the solution of the Rainfall Problem.

• **Knowledge about Bugs**: The tutor uses two types of knowledge about bugs to make the decision not to address the two missing output guard bugs. First, the importance of missing output guard bugs is not very high, for reasons that are discussed in the next section. Second, the effects of the bugs are trivial: The bugs do not have any impact on key actions in the program because they only affect the final output and these effects show up only in rare circumstances. Other missing guards, however, can have important effects. For example, a missing guard against division by 0 in the calculation of AVERAGERAIN can cause a runtime error that can be very confusing to students.

Thus, in this scenario, the tutor uses three different types of knowledge simply to decide whether to tutor bugs. Similar accounts can be given of the dependence of the other four Tutorial Planning Decisions on the types of knowledge presented in Figure 2.4.

In summary, tutors use many different types of knowledge to make the five Tutorial Planning Decisions. The types of knowledge are not random, however. Rather, each type of knowledge can be placed into one of four main categories of knowledge, as this overview has suggested. The next section gives a detailed account of the types of knowledge that fall into the four main categories.

TUTORIAL PLANNING DECISION	KNOWLEDGE CATEGORY			
	Student	Domain	Bugs	Teaching
1. Grouping		plans	patterns	grouping rules
2. Whether	ability	task spec	importance effects	
3. Goals	knowledge motivation	concepts	cause	tutorial goals
4. Ordering			effects	ordering rules
5. Strategy	ability	concepts		strategies

Figure 2.4 : Knowledge Types and the 5 Tutorial Planning Decisions

2.4. Types of Knowledge in the 4 Knowledge Categories

The previous section briefly identified and described the four categories of knowledge that tutors use to make the five Tutorial Planning Decisions. As well, an example was given of how knowledge types in three of the four categories affected a specific tutorial decision, namely whether to tutor two similar, trivial bugs.

The major goal of this section is to describe in more detail the tutorial knowledge types that comprise the four categories. The

presentation is organized around the four categories of knowledge. Each tutorial knowledge type described in a section devoted to its parent category. For example, tutorial knowledge about causes of bugs and tutorial knowledge about effects of bugs are both described in the third subsection, entitled Knowledge about Bugs.

The content of each of the tutorial knowledge types was derived primarily from the Questionnaire Study though some content was generated from analysis of the results of the Real Time Study. For example, analyses of tutors' planning in the Questionnaire Study revealed four primary tutorial strategies that they use with students, including requesting justifications and asking students to hand simulate sections of code. In the Real Time Study, tutors were observed to use the tutorial strategies identified in the Questionnaire Study.

It is important to realize that the characterizations of tutorial knowledge types is not intended to be exhaustive in the sense that the characterizations account for *every* piece of knowledge that *every* tutor *ever* referred to in generating a tutorial plan. Rather, the knowledge identified in the following descriptions is both a) *widespread*, in the sense that most tutors studied empirically used the knowledge, and b) *sufficient* for generating the tutorial plans observed in the empirical studies of human tutors (cf. Littman, Pinto, & Soloway, 1986). The claims that this knowledge is widespread and sufficient are addressed in Littman, Pinto, & Soloway (1989). That paper presents both the empirical findings about the knowledge required to generate tutorial plans and a

description of the statistical methods that were used to analyze the data that support the claims.

2.4.1. Knowledge Category 1: Knowledge about Students

As Figure 2.3 shows, tutors use three types of knowledge about students when they generate their tutorial plans. The first three tutorial knowledge types shown in the figure, student ability, student motivation, and student intentions, are the simplest. The fourth tutorial knowledge type, student knowledge, is somewhat more complex.

- Student Ability: Student Ability has one of three values, low, medium, or high. Tutors use Student Ability primarily to decide what strategy to use with a student. Students who have high ability need less help from the tutor than students who have low ability and a tutor's strategy selection reflects this. Student ability is coded as a three-valued variable because the empirical data reported in Littman, Pinto, & Soloway (1989) suggest that tutors do not make finer distinctions. For example, tutors say such things as "This student doesn't know what's going on.", "This student seems to be doing Okay.", and "This student seems to really know what's going on so I think these are just slips."

- Student Motivation: Student Motivation has one of three values: low, medium, or high. Tutors use Student Motivation primarily to decide how to begin a tutoring session, how long the tutoring session will be, and what strategies are appropriate for the student. For example, if a student has low motivation, tutors

typically begin the tutoring session with an easy problem and give the student enough help so that the student gains confidence in the ability to find, and fix, bugs. Student motivation is treated as a three-valued variable for the same reasons that student ability is treated as a three-valued variable.

- Student Intentions: Student intentions are the goals that the student wants to solve. Each goal that a student solves is part of the task specification or is a transformed, buggy task specification. Each goal that a student solves is either 1) solved unknowingly (a spurious goal) or 2) is solved intentionally. For the most part, students' intentions match the goals in the task specification. This is not always the case, however. For example, a student might put the output statements for the Rainfall Task inside the loop to cause the output to occur on each iteration of the loop. That is, the student intends the values of AVERAGRAIN, MAXIMUM, and so on to be written out on each pass of the loop whereas the Rainfall Task Specification implies that the output statements should *follow* the loop.

- Student Knowledge: Student Knowledge is the most complex of the three tutorial knowledge types in Knowledge Category 1. Tutors reason about a) programming concepts the student understands and b) programming skills the student is able to use effectively. The measure of a student's proficiency is a three-valued attribute with the values high, medium, and low. The concepts and skills that students know about are a type of domain knowledge. They are discussed in the subsection devoted to tutorial knowledge types in the category of Domain Knowledge.

As a brief example of student knowledge, consider an alternative scenario for the tutor trying to decide whether to address BUG 8 and BUG 9 in the program shown in Figure 1.1. Suppose that the student's ability were high but the student was weak on the skill of testing programs for boundary conditions, which could lead to BUG 8 and BUG 9. If the tutor wanted to address the skill of testing programs for boundary conditions -- tutor often do this with good students -- then BUG 8 and BUG 9 provide a good setting for that goal. Thus, in this example, the tutor's knowledge about the student's skill affects the the tutorial goals that the tutor establishes (Tutorial Planning Decision 3) and whether the tutor addresses certain bugs (Tutorial Planning Decision 2).

In summary, tutors use three types of knowledge about students when they generate their tutorial plans. *Student ability* and student motivation are simple three-valued attributes. Student knowledge has two subtypes, programming concepts and programming skills. As will be seen later, each of the types of knowledge about students influences more than one of the five Tutorial Planning Decisions.

2.4.2. Knowledge Category 2: Knowledge about the Domain

Figure 2.3 shows the four tutorial knowledge types about the domain of programming that tutors use to generate tutorial plans. The first of the four knowledge types shown in the figure, task specifications, is specific to solving assignments that students receive in their classes. For example, the Rainfall task

specification requires that the program have variables for RAINFALL, AVERAGERAIN, MAXIMUM, etc.; the Rainfall task specification must have a computation for AVERAGERAIN and output statements for AVERAGERAIN, MAXIMUM, and RAINDAYS. Thus, task specifications are specific to problems. The remaining three types of knowledge about the programming domain are not specific to any assignment. For example, a programming plan for interactive data collection is required by many task specifications; the concept of iteration is ubiquitous in programming; and the skill of testing for boundary cases is used in writing virtually any program. The purpose of this section is to describe the four types of knowledge about the domain of programming that are used in the construction of tutorial plans.

2.4.2.1. Task Specifications

Task specifications are represented in terms of the goals that must be solved to produce a correct program. Goals in the task specification are connected by a precondition relation. For example, the goal of calculating AVERAGERAIN requires the preconditions shown in Figure 2.5:

- the variables that participate in the calculation must be declared
- the variables that contain calculated values must be initialized
- the values of variables from which AVERAGERAIN is derived must be calculated.

Finally, the focus of the goal, the division of TOTALRAIN by TOTALDAYS, is represented as the action of the goal. The action for the goal specifies what should be done (calculate AVERAGERAIN) and plans that can be used to do the calculation. In the case of the calculation of AVERAGERAIN the plan is to divide TOTALRAIN by TOTALDAYS.

Each of the preconditions shown in Figure 2.5 is also represented as a goal which has preconditions attached to it. Tracing all preconditions from all output goals results in a complete task specification. For example, the Rainfall Task Specification has approximately forty goals, including input and calculation, and output guards; an input statement for RAINFALL; an error recovery in case the user enters illegal data; and so forth. The other assignment that is represented in the knowledge base of the tutorial planner, the Reformatting Task Specification, also has approximately forty goals.

In summary, the task specification for a problem is a representation of the goals that must be solved for a correct solution. The goals in the task specification are connected by a precondition relation which makes it possible to determine all dependencies in the task specification.

GOAL:
    Calculate AVERAGERAIN
PRECONDITIONS:
    Declare AVERAGERAIN, TOTALRAIN, TOTALDAYS
    Initialize TOTALRAIN, TOTALDAYS
    Calculate TOTALRAIN, TOTALDAYS
ACTION:
    Calculate AVERAGERAIN
    PLANS FOR ACTION:
        PREFERRED PLAN: Divide TOTALRAIN by TOTALDAYS
        OTHER PLANS: None

Figure 2.5: Preconditions for Calculating AVERAGERAIN

2.4.2.2. Programming Plans

Programming plans are not specific to task specifications. Programming plans are used to solve general goals such as "accumulate each value of an input variable into a summation variable". Using the same programming plan to solve two examples of the same general goal requires only that the variable names in the programming plan be changed to correspond with the task specification. For example, the same programming plan can be used to output the value of AVERAGERAIN and the value of MAXIMUM. The only difference is that the variables accessed by the output statements are different in the two cases.

Approximately ten programming plans are required by the Rainfall task specification and the Reformatting task specification. Most of these are shown in Figure 2.6. These programming plans can

be used to solve many other task specifications as well. Figure 2.6 shows several important plans and gives a description of each. For example, the first plan is a Running Total Input Loop Plan. The goal of the Running Total Input Loop Plan is to create a sum of a sequence of values. The example given shows the variable Totl being augmented by the value of New until New takes on the value Stop: Stop could be declared as a numeric constant such as 99999. The second example shown in Figure 2.6 is a Calculate Plan. There are many Calculation Plans. Each Calculation Plan derives a value for a resultant variable from one or more other values. The example Calculation Plan shown is of the calculation of an average -- the same plan required for the Rainfall Task Specification. The remaining plans shown in the figure are plans that are ubiquitous in programming. An extensive description of programming plans can be found in Spohrer, Pope, Lipman, Sack, Freiman, Littman, Johnson, Soloway, 1985 and Johnson, 1985.

Each programming plan that is used to generate tutorial plans for multiple bugs specifies the name of the plan and alternative ways of implementing it. For example, the Running Total Input Loop Plan can be written with a WHILE construct or a REPEAT-UNTIL construct. Figure 2.7 shows three different ways to write the Running Total Loop Input Plan in the PASCAL programming language.

- The first example using WHILE reads a value for New, enters the loop if the value of New is not Stop, adds the value of New (guaranteed not to be equal to Stop) to Totl, gets another value for New and cycles through the loop again.

- The second example in the column of examples with the WHILE loop first enters the loop, gets a value for New, checks to see that it is not equal to the loop termination value, and then adds the value into Totl.

- The example in the column for the REPEAT-UNTIL loop enters the loop, gets a value for New, checks to be sure that New does not equal Stop, adds New to Totl if New does not equal Stop, and continues this process until New equals Stop.

In short, there are many ways to instantiate the same programming plan. Each programming plan has a default preference ordering its instantiations. For example, the default preference for the instantiations for the Running Total Loop Input Plans shown in Figure 2.7 is 1) the first example in the WHILE column, 2) the second example in the WHILE column, 3) the example in the REPEAT-UNTIL column. Intuitively, the preference ordering is defined according to the way in which the Running Total Loop Input Plan instantiations are taught in novice programming classes. For example, the first WHILE loop instantiation does not duplicate the test for the equality of New and Stop, and obtains a value for New before entering the loop. This instantiation seems to be easiest for students to learn.

Programing Plan	Description	Example

Running Total Input Loop	Sum value sequence in variable Totl	Readln (New);
		While (New <> Stop)
		Begin
		Totl := Totl + New;
		Readln (New)
		End;

Calculate	Calculate value from other values	Ave := Totl/Days;
Guard Calculation	Guard calculation from bad cases	If Days = 0 Then Writeln ('Bad Data');
Input	Get value from user	Readln(New);
Output	Write out value of variable	Writeln(Totl);
Declare	Bring variable into existence	Var Totl : Real;
Initialize	Give variable initial value	Totl := 0;

Figure 2.6: Examples of Important Plans

WHILE Loop	REPEAT-UNTIL Loop	

Readln(New);
While (New <> Stop)
Begin
Totl := Totl + New
Readln(New)
End;
	Repeat
	Readln(New);
	If New <> Stop Then
	Totl := Totl + New;
	Until(New = Stop);	

While (New <> Stop)
Begin
Readln(New);
If New <> Stop Then
Totl := Totl + New;
End;		

Figure 2.7: Three Ways to Write a Running Total Loop Input Plan

2.4.2.3. Programming Concepts

Tutors use a student's knowledge of programming concepts to decide 1) how to help students fix their programs and 2) what they would like teach the student about general programming topics. For example, if a tutor believes that a student misplaced an update that belongs in a loop because of a misconception about scoping, such as BUG 5 in Figure 1.1, then the tutor would attempt to correct the student's concept of scoping as a means to fix the bug. On the other hand, tutors know that iteration is an important concept. Even if the student has not made any bugs that implicate knowledge of iteration, a tutor may work with the student on the iteration concept to extend the student's knowledge.

Figure 2.8 shows the important programming concepts that are involved in tutorial planning for multiple bugs. In the figure, each concept is named, and a description and example of it is given. For example, the first concept in the figure, Selection, refers to the topic of when and how to decide on one of several actions. The example for Selection is the If-Then-Else construct, which is part of virtually all programming languages. The If-Then-Else construct provides a way to choose either of two actions depending upon the value of some test. The last concept in the figure is Typing. Typing is an important programming concept because many languages, such as PASCAL, enforce typing. Typing is the assignment of properties to objects in the program. The properties of an object determine the operations that can be performed on them. For example, if a variable

is a string of characters, then it is not possible to add a number to it; if a variable is an integer, it is not possible to ask for the third character in the integer.

Students either are strong, medium, or weak with respect to a programming concept. Each programming concept has a default importance of either low, medium, or high. The concepts shown in Figure 2.8 are all either high or medium importance. Concepts such as declaration and initialization are low importance.

Concept	Description	Example
Selection	Deciding upon one of many actions	If-Then-Else
Iteration	Repeating the same actions	Repeat-Until
Sequencing	Ordering actions	Control flow
Scoping	Performing actions together	Begin-End block
Input	Getting information from world	Reading from a user
Output	Giving information to the world	Writing to a user
Calculation	Deriving a new value	Find average
Guarding	Preventing unwanted actions	If-Then
Boundary Cases	Handling all possible cases	Zero-division error
Typing	Making objects have properties	Integers

Figure 2.8: Primary Programming Concepts

2.4.2.4. Programming Skills

Figure 2.9 shows the important programming skills that tutors are concerned to help students acquire or improve. Each of the skills has many different subcomponents. For example, the skill of Testing involves

- identifying the part of the program to test
- generating appropriate input
- characterizing the expected output
- collecting the actual output
- comparing the expected and actual output
- attributing discrepancies to their causes.

Equally, the skill of Reorganizing involves determining when to reorganize, what to reorganize, and how to reorganize. For example, if the actions in a loop are guarded against unwanted conditions and the same guard statement appears twice in the loop, this is a clue that the loop should be reorganized.

Each programming skill can be the focus of tutoring but more often the skill is addressed in the context of fixing a bug. For example, the tutor might help the student improve testing skills while helping to fix output boundary guard bugs.

Skill	Description	Example
Testing	Knowing how to test programs	Test boundary cases
Simulating	1. Identify segment to simulate 2. Mentally run program 3. Observe behavior	Simulate Main Loop
Debugging	Finding bugs and fixing causes	Fix Infinite Loop
Comparing	Comparing actual with desired	Verify calculation
Reorganizing	Don't patch -- Redo	Restructuring a loop

Figure 2.9: Primary Programming Skills

2.4.2.5. Summary

Tutors have four main types of knowledge about the domain of programming. The four main types are: knowledge about task specifications, knowledge about programming plans, knowledge about programming concepts, and knowledge about programming skills. Knowledge about task specifications provides a representation of the goals of students' assignments and is therefore specific to particular programming assignments. Knowledge about programming plans, concepts, and skills is general to the domain of programming. Therefore students can use this knowledge to solve any assignment and tutors often emphasize it when they work with students.

2.4.3. Knowledge Category 3: Knowledge about Bugs

Figure 2.3 shows the types of tutorial knowledge about bugs that tutors use when they generate tutorial plans. As the figure shows, tutors use knowledge about causes of bugs, effects of bugs, the importance of bugs, and patterns of bugs. For example:

- tutors know about causes of bugs, such as slips, misconceptions, and missing knowledge
- tutors know about effects of bugs, such as a variable having a wrong value, one program action clobbering another program action before it should
- tutors know about patterns of bugs, such as one bug being created in an attempt to repair another bug
- tutors know about the importance of bugs, which can affect whether and how a bug is tutored.

The purpose of this section is to describe each of the four knowledge types that are in Knowledge Category 3, the category of bugs.

2.4.3.1. Causes

This section gives the details of the causes that tutors reason about when they generate tutorial plans. Bug causes have two components, a cause *type* and a cause *object*. For example, a student might have a misconception (the cause type) about the scope of loops (the cause object, one of the programming concepts discussed in the previous section). Indeed, this misconception may have led the student who

wrote the program in Figure 1.1 to make BUG 5, the misplaced update of TOTALRAIN below the main loop.

The *criticality* of a cause is a function of the seriousness of the cause type and the importance of the cause object. For example, students frequently do not know a plan that is required to solve an assignment. The default seriousness for the cause type 'not knowing' is high. The criticality of the cause, however, depends upon the importance of the object of the cause. For example, if a student does not know the running total plan, an important plan in procedural programming languages, the criticality of the cause is high. If a student does not know about a less important plan, such as guarding output statements against the case of no input, the criticality is low because of the low importance of guarding output against the case of no input. Thus, a simple function of cause type and cause object determines the criticality of a cause.

Figure 2.10 identifies each cause type in the first column, gives an underlined definition and an example in the second column, and displays the default seriousness measure in the third column. The cause type of a bug can range from a slip, which occurs when the student neglects to do something that the student knows to do; to a misconception which means that the student has a mistaken idea about some concept that leads to an incorrect solution; to failure to test, which means that the student did not perform sufficient testing to find the bug. As well, the default seriousness of each cause type is indicated. For example, a misconception has a high seriousness rating and is more serious than a slip which has a low seriousness rating.

Each cause object is in one of the four tutorial knowledge types in the second knowledge category shown in Figure 2.3, Domain Knowledge; these tutorial knowledge types are discussed in an earlier section. More examples of combinations of cause types and cause objects are given in the section which discusses the rules that govern the five Tutorial Planning Decisions.

In summary, bug causes are made up of two components, a type, such as a *misconception* or *forgetting* to perform an action, and an object, such as the *concept of scoping of a loop*, or *plans for accumulating a running total* such as TOTALRAIN. Each cause type has an associated seriousness and each cause object has an importance measure associated with it from which the criticality of the cause is calculated The criticality of a cause is thus a simple joint measure of the seriousness of the cause type and the importance of the cause object.

CAUSE TYPE	DEFINITION & EXAMPLE	SERIOUSNESS
slip	Student formed intention but did not carry out forgetting to initialize variable	LOW
overlook	Student never formed intention missing output of RainyDays	LOW
misinterpret goal	Student misinterpreted a goal calculate AverageRain with RainyDays	HIGH
rationalized away	Student convinced that required action unnecessary	HIGH
not know	Student did not know required piece of knowledge not knowing Process/Read loop	HIGH
uncertainty about	Student unsure about concept, conditions for action etc. not sure when to initialize variables	MEDIUM
misconception	Student has misconception belief that all numeric variables must be same type	HIGH
delay development	Student has intent to solve goal, just hasn't done it yet not create code to output maximum until rest works	LOW
version hacking	Student created error because of extensive patching misplaced code caused by moving code around	MEDIUM
try optimization	Student tried to solve goal in a better way & created bug	MEDIUM
attempted repair	Student tried to fix one mistake and created another initializing divisor in AverageRain to 1	MEDIUM
plan pollution	Student solved problem using previous, wrong method create next value RainDays by adding 1 to prior value	LOW
failure to test	Student did not test program in way to uncover bug not see missing guard on calculation of AverageRain	LOW

Figure 2.10: Causes of Bugs

2.4.3.2. Effects

A bug effect is defined as the impact of a bug on the behavior of a program. Bug effect types range from a variable having a wrong value to the inconvenience a user experiences if the output of a program is not labelled. There are approximately 10 effect types that tutors consistently use when they generate tutorial plans. The

purpose of this section is to identify the different types of effects that bugs can have on the behavior of a program.

For example, if a student initializes the counter variable RAINDAYS to 1, perhaps in an attempt to avoid division by zero in the calculation of the average, then there are several effects of the bug:

- an *off by one* value for RAINDAYS
- a *wrong value* for AVERAGERAIN
- a *wrong value output* AVERAGERAIN

This example therefore shows that a single bug can have several effects.

Some effects of bugs are easy for most students to find. For example, *no output* for AVERAGERAIN would be easy for most students to find. Other bugs are difficult for most students to find. For example, the effect of a misplaced statement that clobbered only the first value of a variable being added into a running total would be difficult for most students to find. BUG 1 in Figure 1.1 is an example of this sort of bug.

Figure 2.11 shows the effects of bugs, their seriousness, and the likelihood that most students can find them. For example, a wrong value for a variable has a medium seriousness whereas a missing value (no value) is more serious. These two effects have different likelihoods that students can find them; no value for a variable is very salient whereas a wrong value can only be detected if the student performs hand simulation of the program and thereby anticipates the values that variables should take on.

EFFECT TYPE	SERIOUSNESS	FINDABILITY	
Wrong Value	MEDIUM	LOW	
No Value	HIGH	HIGH	
Calculation Error	MEDIUM	MEDIUM	
Off By One	LOW	LOW	
Infinite Loop	HIGH	HIGH	
Wrong Output	MEDIUM	MEDIUM	
No Output	HIGH	HIGH	
Clobber Required Value	MEDIUM	LOW	
User Inconvenience	LOW	LOW	

Figure 2.11: Effects of Bugs

2.4.3.3. Types

Bug types provide a classification scheme for individual bugs, in contrast to the bug patterns described in the next chapter, which characterize relations among several bugs. For example, BUG 4 in the program shown in Figure 1.1 is a *missing* statement to read a new value of RAINFALL on each iteration of the main processing loop; BUG 5 is a *misplaced* update of TOTALRAIN; BUG 7 has the *wrong* variable in the output statement for the number of rainy days. The italicized words in the previous sentence are types of bugs and this section identifies and describes the six types of bugs that are used in generating a tutorial plan.

The typology of bugs used for tutorial planning is similar to that described in Johnson, Soloway, Cutler, and Draper, 1983. Figure 2.12 shows the typology of individual bugs. There are six types of bugs, as the figure shows. Each of the six types named in the figure is defined by an underlined statement and an example of that type is given just below.

BUG TYPE	DEFINITION & EXAMPLE
Missing	The code is missing No counter for RAINDAYS
Misplaced	The code is in the wrong place Calculating AVERAGERAIN in the loop
Spurious	The code is not required by the task Maintaining a counter DRYDAYS
Wrong	The plan of the student is incorrect WHILE not IF for simple conditional
Malformed	The plan of the student is imperfect Adding 1 to update TOTALRAIN
Inelegant	There is a better way to achieve task goal Subtract sentinel from running total

Figure 2.12: Typology of Individual Bugs

2.4.3.4. Plan Location

The location of a programming plan is specified relative to other plans in the program. For example, the misplaced update of TOTALRAIN shown as BUG 5 in Figure 1.1 is AFTER the main loop and BEFORE the calculation of AVERAGERAIN. The descriptors for plan location are intuitively obvious and consist of BEFORE, INSIDE, and AFTER. The description of the correct location of a plan is contained in the representation of the task specification, discussed above. The actual location of the student's buggy plan is a feature of the description of the student's bugs. Thus, the tutor knows where a plan should be and where the student put it. For example, the update of TOTALRAIN should be INSIDE the main loop; the student, however,

put the update of TOTALRAIN AFTER the main loop and BEFORE the calculation of AVERAGERAIN.

2.4.3.5. Summary

Tutors have four types of knowledge about bugs. Tutors reason about the causes of bugs, the effects of bugs, types of bugs, and the location of bugs. Each of these four types of knowledge about bugs affects tutorial planning. For example, the cause of a bug can affect the tutorial strategy the tutor uses to help the student with the bug; a bug which has serious effects on a program, such as an infinite loop, is more likely to be addressed than a bug the has minor effects, such as the wrong value output in an obscure boundary condition.

2.5. Chapter Summary

In summary, this chapter has identified four categories of knowledge that tutors use to generate a tutorial plan. Within each of the four categories there are several types of knowledge. For example, in the category of knowledge about bugs, there are four types of knowledge -- knowledge about causes, effects, types, and location.

Each of the types of knowledge in the four categories plays a role in making one or more of the five Tutorial Planning Decisions which are necessary to create a tutorial plan. For example, tutors

must decide which bugs are worth addressing and which bugs are not worth working on. A bug can be eliminated from the tutorial plan for several reasons. A bug may be too hard for the student to understand in which case the tutorial plan will not address it. The bug may be too trivial and therefore take up valuable time needed for other, more important bugs.

This chapter has provided detailed descriptions of three of the four categories of knowledge, namely knowledge about students, knowledge about the domain, and knowledge about bugs. The next chapter describes in detail the fourth category of knowledge: knowledge about teaching, which includes knowledge about tutorial planning and knowledge about executing tutorial plans.

Chapter 3

KNOWLEDGE CATEGORY 4: KNOWLEDGE ABOUT TEACHING

3.1. Introduction

Tutors have two primary types of knowledge about teaching. First, tutors have knowledge that they use to construct their tutorial plans. For example, tutors know which bugs are important to tutor and which bugs do not matter. Second, tutors have knowledge about how to execute tutorial plans. One type of knowledge about tutorial plan execution is how to recover from failures during tutoring. For example, if a tutor makes an assumption about why the student made a bug that later turns out to be incorrect, tutors know how to modify their tutoring to incorporate the correct cause. Knowledge about how to construct tutorial plans and knowledge about how to execute tutorial plans are the two types of knowledge in Category 4, knowledge about teaching, as shown in Figure 2.3.

The purpose of this chapter is to describe the knowledge that tutors use to generate tutorial plans -- knowledge about executing tutorial plans and recovering from failures is discussed in Chapter 7. As noted in Chapter 2, all tutors do not generate exactly the same tutorial plans. As for Chapter 2, the goal of this chapter is not to

account for individual differences among tutors but, rather, to describe the predominant decisions that tutors make as described in Littman, Pinto, & Soloway, 1989.

3.2. Tutorial Planning Knowledge

A major claim of the dissertation is that experienced tutors construct a tutorial plan when they must help a student who has made multiple bugs and that the construction of the tutorial plan revolves around the five Tutorial Planning Decisions that were identified in the first chapter of the dissertation and are displayed in Figure 1.6. The goal of this section is to describe the knowledge that tutors use to make the five tutorial planning decisions.

The section is divided into five subsections, each devoted to one of the five Tutorial Planning Decisions. Each of the five subsections has the same format. First, the tutorial planning decision is described and a brief example is given; next, the conceptual knowledge is identified that the tutorial planning decision depends on; finally, the rules are described that use the conceptual knowledge to make the tutorial planning decision and examples of the rules are given.

3.2.1. Tutorial Planning Decision 1: GROUPING

The result of the Grouping Decision is that all the student's bugs are sorted into groups which have one or more bugs. For example, we

have seen that BUG 8 and BUG 9 in Figure 1.1 are placed in the same group by tutors because both bugs are in the same plan component, namely guards for output statements. In essence, the grouping decision is a classification problem in which the goal is to put bugs together that are somehow "related". The goal of this section is to describe what is meant by the term "related". The remainder of this section describes the bug groupings and illustrates the bug groupings with examples, and then describes the rules that sort bugs into groupings.

3.2.1.1. Bug Groupings and Subgroupings

Figure 3.1 displays the conceptual tree of bug groupings that underlies the classification of multiple bugs. The first level in the tree contains six groupings, some of which have subgroupings. For example, the third grouping from the left is labeled "Same Plan Component"; BUG 8 and BUG 9 in Figure 1.1 are in the offspring grouping "Housekeeping" of the Same Plan Component grouping. BUG 8 and BUG 9 are in the Same Plan Component grouping because the bugs are both in the same plan component; they are in the Housekeeping grouping because the output guard plan component is a minor plan component. The six main groupings and their subgroupings, illustrated by examples are:

• Bug Dependency Bug Dependency groupings relate bugs in which one of the bugs in the grouping results because of other bugs in the grouping. For example, if a student initializes the divisor TotalDays to 1, a bug, in order to avoid a division by zero in the

calculation of AverageRain, another bug, then the student has created the first bug because of the second. In this case the student has attempted to repair the division by zero bug by introducing the incorrect initialization bug. In terms of the conceptual tree of bug groupings, the incorrect initialization bug and the missing guard on the AVERAGERAIN calculation, BUG 1 and BUG 2 in Figure 3.2, is in the grouping *Bad Repair,* a subgrouping of *Program Behavior Interaction,* which is a subgrouping of *Bug Dependency.* An example of a *Good Repair* grouping is shown in Figure 3.3. Here the student has allowed the sentinel value that is intended to signal the end of input to be added into TOTALRAIN. The student's repair to this bug is to subtract the sentinel from TOTALRAIN just before calculating AVERAGERAIN. Although the student's repair is correct, it is inelegant and nearly all human tutors group the two bugs together.

The two remaining Bug Dependency subgroupings are Undetected Program Behavior Interactions and Consistency Maintenance Program Behavior Interactions. Undetected Program Behavior Interactions occur when the program contains interactions between program components and the student does not know about the interactions. For example, BUG 1 in Figure 1.1 is a case of an Undetected Program Behavior Interaction. The assignment of 0 to the variable RAINFALL destroys the first value of RAINFALL acquired in the READ(RAINFALL) statement just above the buggy assignment. Thus, it seems that the student did not know about the bug.

Consistency Maintenance Program Behavior Interactions occur when the student creates a second bug to maintain consistency with another bug. For example, in Figure 3.4 the student has made two

bugs. The first bug is the declaration of all variables to be integers. The second bug results from the first: The student knows that dividing integers requires the DIV operation and so uses DIV in the calculation of AVERAGERAIN. The use of DIV results from the student's desire to maintain consistency between the improper variable declarations and the calculation of AVERAGERAIN.

In short, Bug Dependency groupings are those in which there is some sort of "dependency" between bugs.

• Same Plan Same Plan bug groupings are those in which all the bugs are in the same parent plan. For example, if the student forgets to initialize the variable MAXRAIN and also forgets to include an output statement for it, then these two bugs would be put in a Same Plan grouping. If the plan is an important plan, such as the calculation of AVERAGERAIN, then it is a Main Same Plan grouping. The missing initialization and output statements for MAXRAIN are a Main Same Plan grouping. If the plan is not an important plan then the grouping is a Secondary Same Plan grouping.

• Same Plan Component Same Plan Component bug groupings are those in which all the bugs are in the same plan component. For example, the two missing output guards, BUG 8 and BUG 9 in Figure 1.1, are both in a Same Plan Component grouping. These two bugs are in a Housekeeping subgrouping because they are unimportant plan components.

If there are several bugs in the same plan component, say an input statement, and the parent plan is focal -- i. e., very important in the solution of the main goals of the task specification, then the bugs would be in the Focal subgrouping of Same Plan Component. If

some of the bugs were in focal parent plans and some were in unimportant parent plans, then the bugs would be placed in a Mixed subgrouping of Same Plan Component.

• Diagnosis If the tutor does not know exactly why the student made one or more bugs, and the tutor wants to determine the reason, then the bugs are placed in a bug grouping of type Diagnosis. If the tutor has no strong hypothesis about the cause of the bugs, as is the case for most tutors and Bugs 2 through 5 in Figure 3.5, the strange updates, then the bugs are placed into the subgrouping Exploratory Diagnosis.

If the tutor has a good idea of the cause of a bug, but has a competing hypothesis, then the tutor would place the bugs in the subgrouping Confirmatory Diagnosis.

• Hard Hard bug groupings are those in which a key bug, one that is very important but hard to tutor, is the focus of tutoring and one or more other bugs, the supporting bugs, are treated before the key bug expressly to facilitate tutoring the key bug.

The two kinds of supporting bugs are clean-up and lead-in bugs. A clean-up bug is one which, if eliminated, makes the parts of the program affecting the key bug easier for the student to understand. A lead-in bug is one whose tutoring gives the student useful information or practice with skills that are needed for tutoring the key bug.

For example, in Figure 3.6, the key bug is BUG 4, the update of TOTALRAIN which is misplaced below the loop. BUG 3, the missing input statement for RAINFALL, causes an infinite loop, which students find confusing. BUG 2, the incorrectly formed counter for

RAINDAYS, is a very easy bug which the tutor can use to get the student thinking about what happens each time the loop executes. Because the student has a misconception about the scoping of loops, and therefore incorrectly believes that the update of TOTALRAIN occurs on each execution of the loop, this is exactly what the tutor wants to focus on for BUG 4, the misplaced update of TOTALRAIN. However, before getting the student to observe or simulate the behavior of the loop, it would be a good idea to eliminate the infinite loop. This reasoning leads to a Hard grouping in which

- BUG 3, the missing input statement for RAINFALL is addressed first; this is a clean-up bug
- followed by BUG 2, the unconditional counter for RAINDAYS, which is a lead-in bug
- finally addressing the key bug, the misplaced update for TOTALRAIN.
- Singleton Singleton groupings are groupings with only one bug. The bugs in Singleton groupings are those for which no good case can be made to place them in a grouping with other bugs. For example, a student might assign 9999 to SENTINEL, the constant for a loop termination variable. Because there are no other related bugs, this bug would appear in a Singleton grouping in the tutorial plan.

In summary, bugs are classified into groups according to relations with other bugs. For example, two missing output guards are related because they are in the same plan component; a bug created in the attempt to fix another bug are related by a Repair

relationship; bugs which facilitate tutoring a key bug may be placed in a Hard grouping.

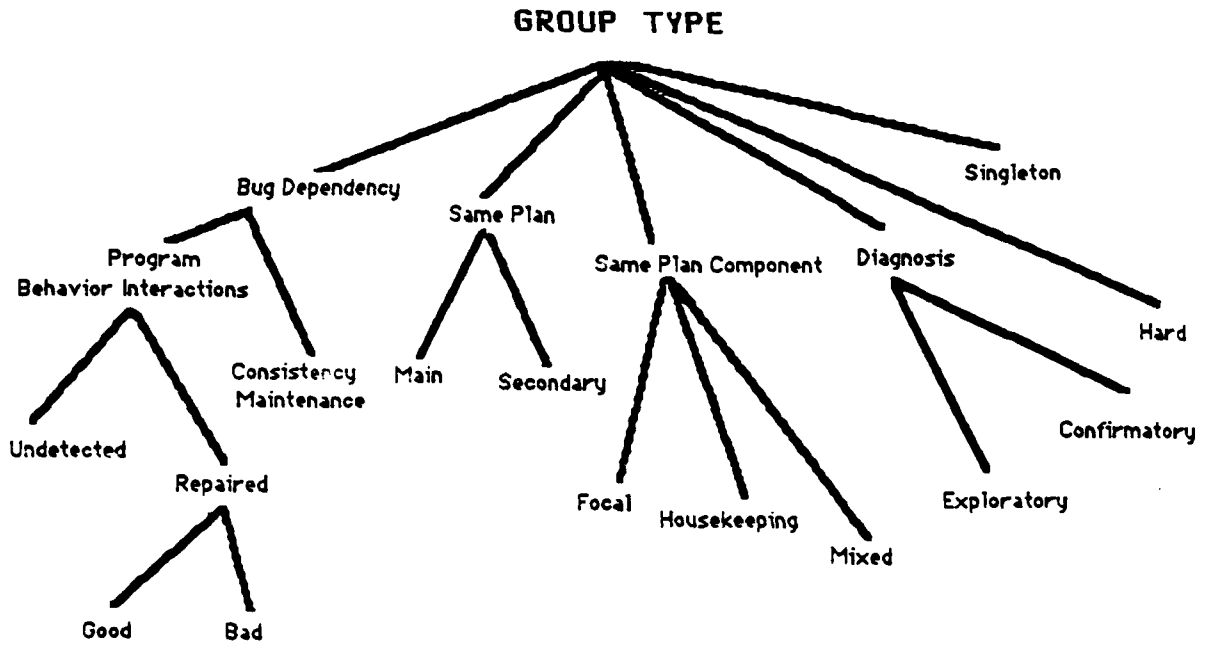


Figure 3.1: Group Type

3.2.1.2. Bug Grouping Rules

The purpose of this section is to describe the grouping rules that decide how to place bugs in groups in the tutorial plan, Tutorial Planning Decision 1. Because most bugs in a student's program could be placed in any of several groups, but some groupings are preferable to others (e. g., a bug should be placed in a Singleton group only as a last resort), this section 1) illustrates the rules that are used to

make bug groups and 2) describes the priority scheme that ensures that bugs are placed into preferred groups.

The rules that make Tutorial Planning Decision 1 can be thought of as recognizers for the groupings and subgroupings shown in Figure 3.1. Each grouping, such as the Same Plan Component grouping, is based on combinations of features of bugs. The recognition rules corresponding to a bug grouping identify the legal combinations of features for the grouping. For example, BUG 8 and BUG 9 in Figure 1.1 both are in the same housekeeping plan component, the output guard. Thus, when these two bugs are encountered by the rules that recognize the Same Housekeeping Plan Component grouping, a Same Housekeeping Plan Component bug group is created for the two bugs. In fact, each grouping has several rules that correspond to it. The rules make distinctions between subgroupings and handle housekeeping chores such as adding a single bug to an existing group.

Grouping rules are all IF-THEN rules. The IF clause specifies the conditions under which the THEN clause acts. As an example, consider the following grouping rule, one that creates Bad Repair bug grouping.

GROUP-BUGS:BAD-REPAIR-RULE1

IF **BUG-Y is an incorrect repair for BUG-X**
AND both bugs are unassigned to groups

THEN **Create a bug group of type BAD-REPAIR**
Assign BUG-Y to repairs set
Assign BUG-X to cause set

Such a grouping would be used for the incorrect initialization of TOTALDAYS to 1 and a missing guard in the calculation of AVERAGERAIN, shown as BUG 1 and BUG 2 in Figure 3.2.

The IF part of this rule requires that 1) BUG-Y be an incorrect repair for BUG-X, as BUG 1 is an incorrect repair for BUG 2 in Figure 3.2, and 2) neither bug be assigned to another group. The THEN part of this rule performs three actions:

- Create a BAD-REPAIR bug group, which has two sets of bugs, a set of bugs that are the repairs and a set of bugs that are the cause of the repair.
- Assign BUG-Y to the repair set.
- Assign BUG-X to the cause set.

In short, this rule says that the tutorial planner should 1) place BUG 1 and BUG 2 into the same group, 2) distinguish them according to which bug is a repair (BUG 1) and which bug caused the student to make the repair (BUG 2), and 3) make the group type BAD-REPAIR because the repair, initializing TOTALDAYS to 1, is incorrect.

Figure 3.7 gives example rules for three of the bug groupings and subgroupings shown in Figure 3.1. The rule discussed above, **GROUP-BUGS:REPAIR-RULE1**, has a counterpart that creates bug groups in which repairs are successful: This is the first rule shown in the figure. The second rule creates a bug group for bugs which all are in the same focal plan. As described in an earlier example, if the student left out the initialization and the output statement for the

variable MAXRAIN, then these two bugs would be placed in the same group which would be Same Focal Plan grouping.

The third group type in Figure 3.7 shows three rules that create a Hard bug group. As we have seen, BUG 2, BUG 3, and BUG 4 in Figure 3.6 would be placed in the same Hard group. BUG 4, the misplaced update of TOTALRAIN, would be identified by the first of the three rules, RULE FOR KEY BUG, as the key bug. BUG 3, the missing input statement for RAINFALL, would be identified by RULE FOR CLEAN UP BUGS as a bug that could interfere with tutoring the key bug, BUG 4 because of its effect on the behavior of the program. BUG 3 would therefore be assigned to the clean-up bugs for BUG 4. Finally, BUG 2 would be identified by RULE FOR LEAD-IN BUGS as one that would be useful for tutoring the key bug because BUG 2 would 1) be easy to tutor, 2) focus the student's attention on the behavior of the loop, and 3) lead the student to do hand simulation of the main loop in the program. Thus, BUG 2 would be assigned to the lead-in bugs for BUG 4.

Bugs can be grouped in several ways. For example, a missing initialization of the variable MAXRAIN could be grouped with other missing initializations or with a missing output statement for MAXRAIN. On both empirical grounds, derived from studies of tutors, and functional grounds, and from arguments about good tutoring practice, grouping the missing initialization of MAXRAIN with the missing output statement for MAXRAIN is preferable.

Because bugs can be grouped in several ways, tutors assign priorities to groupings. For example, a repair bug should be grouped with its corresponding bug rather than with other bugs that are in

the same plan component; any bug should be grouped with other bugs rather than treated as a Singleton; a bug which requires diagnosis should be placed in a group of bugs which the tutor suspects all have the same cause, as with BUG 2, BUG 3, BUG 4, and BUG 5, the malformed updates shown in Figure 1.1.

Figure 3.8 shows the priorities which tutors assign to groupings. The priorities derive from analysis of the empirical grouping behavior of tutors in the Questionnaire Study (Littman, Pinto, & Soloway, 1988). The priorities are hierarchical and correspond to the tree structure of groupings shown in Figure 3.1. For example, the most important grouping is Diagnosis: If a tutor does not know why a student made a bug, then it is crucial to determine the cause. Bug Dependencies, as shown in the Figure 3.8 are next in priority. The numbering scheme shows that in the grouping of Program Behavior Interactions, Undetected bugs have a higher priority than Repaired bugs. The remainder of the Figure displays the rest of the priority scheme, with Singleton being the lowest priority.

In summary, there is a conceptual framework for the groupings of bugs that are created for a tutorial plan. The framework is shown in Figure 3.1. Rules identify the criterial features for each grouping e. g., Same Housekeeping Plan Component. A priority scheme controls the preferences given to the different groupings so that any particular bug is placed in the most preferred grouping.

```
PROGRAM RAINFALL (INPUT , OUTPUT);
 CONST
    SENTINEL = 99999;
 VAR
   DAILYRAINFALL, MAXIMUM, AVERAGERAIN: REAL;
   TOTALDAYS,  RAINDAYS : INTEGER;
 BEGIN
BUG 1:  TOTALDAYS SHOULD BE INITIALIZED TO 0
        TOTALDAYS := 1;
        WRITELN ('ENTER DAILY RAINFALL');
   READLN;
   READLN (DAILYRAINFALL);
   WHILE NUM <> SENTINEL DO
    BEGIN
        IF (DAILYRAINFALL >  0) THEN BEGIN

                                .

                                .

                                .
        READLN (DAILYRAINFALL);
     END;
BUG 2:  MISSING GUARD AGAINST TOTALDAYS = 0
      AVERAGERAIN := TOTALRAIN / TOTALDAYS;

                                .

                                .

                                .
   END.
```

Figure 3.2: An Undetected Bug

```
PROGRAM RAINFALL (INPUT , OUTPUT);
 CONST
    SENTINEL = 99999;
 VAR
   DAILYRAINFALL, MAXIMUM, AVERAGERAIN: REAL;
   TOTALDAYS,  RAINDAYS : INTEGER;
 BEGIN
       TOTALDAYS := 1;
  WHILE NUM <> SENTINEL DO
   BEGIN
            WRITELN ('ENTER DAILY RAINFALL');
            READLN (DAILYRAINFALL);
            IF (DAILYRAINFALL > 0) THEN BEGIN
BUG 1:  SENTINEL ADDED INTO TOTALRAIN
            TOTALRAIN := TOTALRAIN + DAILYRAINFALL;
                          .
                          .
                          .
            READLN (DAILYRAINFALL);
     END;
BUG 2:  SUBTRACT OUT SENTINEL FROM TOTALRAIN
      AVERAGERAIN := (TOTALRAIN - 99999) / TOTALDAYS;
                          .
                          .
                          .
     END.
```

Figure 3.3: A Detected Bug

```
PROGRAM RAINFALL (INPUT , OUTPUT);
 CONST
    SENTINEL = 99999;
 VAR
 BUG 1:  ALL VARIABLES DECLARED INTEGER TYPE
      DAILYRAINFALL, MAXIMUM, AVERAGERAIN,
     TOTALDAYS,  RAINDAYS : INTEGER;
 BEGIN
       TOTALDAYS := 0;
       WRITELN ('ENTER DAILY RAINFALL');
       READLN (DAILYRAINFALL);
  WHILE NUM <> SENTINEL DO
   BEGIN
            IF (DAILYRAINFALL >  0) THEN BEGIN
            TOTALRAIN := TOTALRAIN + DAILYRAINFALL;
                              .
                              .
                              .
            WRITELN ('ENTER DAILY RAINFALL');
            READLN (DAILYRAINFALL);
    END;
BUG 2:  WRONG OPERATOR IN DIVISION
      AVERAGERAIN := TOTALRAIN DIV TOTALDAYS;
                              .
                              .
                              .
    END.
```

Figure 3.4: Maintaining Consistency

```
PROGRAM RAINFALL (INPUT , OUTPUT);
  CONST
     SENTINEL = 9999;
  VAR
    NUM, TOT, GREATEST, AVERAGE : REAL;
    SUM, TOTAL, GREAT, RAIN, COUNT, RAINDAYS : INTEGER;
  BEGIN
    WRITELN ('THIS PROGRAM CALCULATES AVERAGE RAINFALL');
    WRITELN ('IT ALSO COMPUTES NUMBER OF RAINDAYS, NUMBER OF DAYS');
    WRITELN ('THAT IT ACTUALLY RAINED AND THE GREATEST RAIN FALL.');
     WRITELN ('ENTER RAINFALL');
    READLN;
     READ (NUM);
     TOT := 0; COUNT :=     GREAT := 0; RAIN := 0;
BUG 1: wrong sentinel test (should be 99999, not 9999)
    WHILE NUM <> SENTINEL DO
     BEGIN
       IF (NUM < 0.0)
      THEN  BEGIN
        WRITELN ('THE VALUE ENTERED DOES NOT MAKE ANY SENSE.');
        WRITELN ('PLEASE CHECK YOUR INPUT AND TRY AGAIN');
       END
      ELSE  BEGIN
BUG 2: malformed update of TOTAL
         IF TOTAL = TOT + NUM
          THEN TOT := TOTAL;
BUG 3: malformed update of SUM
          IF SUM = COUNT + 1
          THEN COUNT := SUM;
BUG 4: malformed update of GREATEST
          IF NUM > GREAT
          THEN GREATEST := NUM;
BUG 5: malformed update inside Boolean expression
           IF ((NUM > 0) AND (RAINDAYS = RAIN + 1))
          THEN RAIN := RAINDAYS;
BUG 6: AVERAGE calculated inside loop
    AVERAGE := TOT/COUNT
BUG 7: no READ(NUM); in loop
          END; END;
    WRITELN (COUNT :0, 'VALID RAIN FALLS WERE ENTERED');
BUG 8: no guard for undefined average
    WRITELN ('THE AVERAGE RAINFALL WAS ',
     WRITELN (AVERAGE :0, 'INCHES PER DAY');
BUG 9: no guard for undefined maximum
    WRITELN ('THE HIGHEST RAINFALL WAS ', GREATEST :2, 'INCHES');
    WRITELN ('THERE WERE ', RAIN :2, 'RAINY DAYS IN THIS PERIOD')
  END.
```

Figure 3.5: Buggy Rainfall Program

```
SENTINEL = 99999;
VAR
RAINFALL, AVERAGE, RAINDAYS, TOTALRAIN : REAL;
BEGIN
(* PROMPT FOR  AND READ IN FIRST VALUE *)
WRITELN ('PLEASE TYPE IN THE FIRST VALUE');
READLN;
READ (RAINFALL);
(* INITIALIZE THE VARIABLE *)
RAINFALL := 0;  HIGHEST := 0;
BUG 1: Missing Initialization of RAINDAYS
(* READ IN NUMBERS UNTIL 99999 IS READ *)
(* ENTER DATA IN ONE DAY AT A TIME *)
WHILE RAINFALL <> 99999 DO
BEGIN
(* CHECK FOR HIGHEST RAINFALL *)
IF RAINFALL > HIGHEST THEN
HIGHEST := RAINFALL;
(* INCREMENT COUNT VARIABLES  DEPENDING ON RAINFALL VALUE *)
IF RAINFALL < 0 THEN
WRITELN ('ENTER ONLY POSITIVE NUMBERS')
ELSE
BUG 2: No Counter For Rainy Days -- RAINDAYS Increments Each Time
RAINDAYS :=  RAINDAYS + 1;
BUG 3: No READ(RAINFALL) In Loop
END;
BUG 4:  Update of TOTALRAIN Below Loop
BEGIN
TOTALRAIN := RAINFALL + TOTALRAIN;
END;
(* COMPUTE THE AVERAGE RAINFALL *)
BEGIN
AVERAGE := TOTALRAIN/RAINDAYS;
END;
(* PRINT OUT THE RESULTS *)
WRITELN;
WRITELN('THE PROGRAM READ IN ', RAINDAYS :0:2, 'RAINY DAYS');
WRITELN('THERE WERE  ', RAINFALL :0:2, 'RAINY DAYS IN PERIOD');
BUG 5: No Guard for Undefined Average
WRITELN('THE AVERAGE WAS  ',  AVERAGE :0:2, 'INCHES PER DAY');
BUG 6: No Guard for Undefined Maximum
WRITELN('THE MAXIMUM WAS  ',  HIGHEST :0:2, 'INCHES');
```

Figure 3.6: A Simplified Buggy Rainfall Program

GROUP TYPE	EXAMPLE GROUPING RULES
GOOD REPAIR	IF BUG-Y is a correct repair for BUG-X AND both bugs are unassigned THEN Create a bug group of type GOOD-REPAIR Assign BUG-Y to repairs set Assign BUG-X to cause set
SAME MAIN PLAN	IF BUG-X and BUG-Y are in the same parent plan AND the parent plan is for an important goal AND both bugs are unassigned THEN Create a bug group of type SAME-MAIN-PLAN Assign BUG-X and BUG-Y to group
HARD	<u>RULE FOR KEY BUG</u> IF BUG-X is important and hard to tutor THEN Create a bug group of type MULTI-HARD Assign BUG-X to key bug <u>RULE FOR CLEAN-UP BUGS</u> IF BUG-X interferes with tutoring KEY-BUG-Y THEN Assign BUG-X to clean-up bugs for KEY-BUG-Y <u>RULE FOR LEAD-IN BUGS</u> IF Tutoring BUG-X focuses attention on either 1) program behavior or 2) concepts useful for tutoring KEY-BUG-Y AND BUG-X is easy to tutor THEN Assign BUG-X to lead-in bugs for KEY-BUG-Y

Figure 3.7: Example Rules Tutorial Planning Decision 1 -- Grouping

3.2.1.1. Tutorial Planning Decision 2: YES-NO

The result of the Yes-No Decision is that all bugs are labeled according to whether or not they will be addressed in the tutoring session. For example, BUG 8 and BUG 9, placed in the same group as described in the previous section, would not be addressed by most

tutors because the bugs do not affect the functioning of the program nor are they key aspects of the task specification. BUG 8 and BUG 9 might be addressed if the student were smart and motivated and there were no serious bugs in the program.

The goal of this section is to describe the three decisions that tutors can make about whether to tutor a bug and to describe the knowledge on which the decisions are based. The YES-NO decision is based on two kinds features, *primitive* features such as the plan component in which the bug occurs, and *computed* features, such as the difficulty of tutoring a bug, which is computed from primitive features of students and bugs. The remainder of this section describes the three YES-NO decisions, gives examples of YES-NO decisions, shows and describes an example rule, and describes the two different kinds of features that are used by the rules that make the YES-NO decision.

Figure 3.9 shows the possible Yes-No decisions that tutors can make when they construct a tutorial plan. The most clear-cut decisions are represented by Yes Definitely, the leftmost leaf in the tree, and No, the third leaf from the left. The two remaining decisions are Yes Try and If Opportunity Arises. If the tutor decides definitely to address a bug, then it is given priority when the tutor makes Tutorial Planning Decision 3, the ordering decision; if the tutor decides not to address a bug, then it is eliminated from the remaining Tutorial Planning Decisions. For example, no tutorial objectives (Tutorial Planning Decision 4) are generated for bugs that the tutor decides not to address.

In the program shown in Figure 1.1, the tutor decides definitely to address BUG 5, the misplaced update of TOTALRAIN. The tutor makes this decision because the cause of BUG 5 is potentially a serious misconception. Equally, the tutor decides not to address BUG 8 and BUG 9, the two missing output guards because, as noted earlier, they are not important for the specification and they do not suggest any deep problems in the student's understanding. The commitments to address the other bugs in the program fall between the two extremes. For example, most tutors are not firm one way or another about addressing BUG 6, the missing guard to prevent division by 0 in the calculation of AVERAGERAIN. This bug can be used to illustrate some lessons about testing the program at boundary conditions, but addressing the misplaced update of TOTALRAIN could potentially consume the entire tutoring session. Thus, the tutor decides to address BUG 6 if the opportunity arises. BUG 1 is more important than BUG 6 because the assignment of 0 to RAINFALL destroys the initial value read by the input statement. This problem is both serious and probably easy to correct, because the initialization is a spurious statement that only needs to be eliminated. A tutor might therefore decide to try to address BUG 1.

1. DIAGNOSIS
    1. CONFIRMATORY
    2. EXPLORATORY

2. BUG DEPENDENCY
    1. PROGRAM BEHAVIOR INTERACTIONS
        1. UNDETECTED
        2. REPAIRED
            1. BAD
            2. GOOD
    2. CONSISTENCY MAINTENANCE

3. SAME PLAN
    1. PRIMARY
    2. SECONDARY

4. HARD

5. SAME PLAN COMPONENT
    1. FOCAL
    2. HOUSEKEEPING
    3. MIXED

6. SINGLETON

Figure 3.8: Priorities for Placing Bugs in Groups

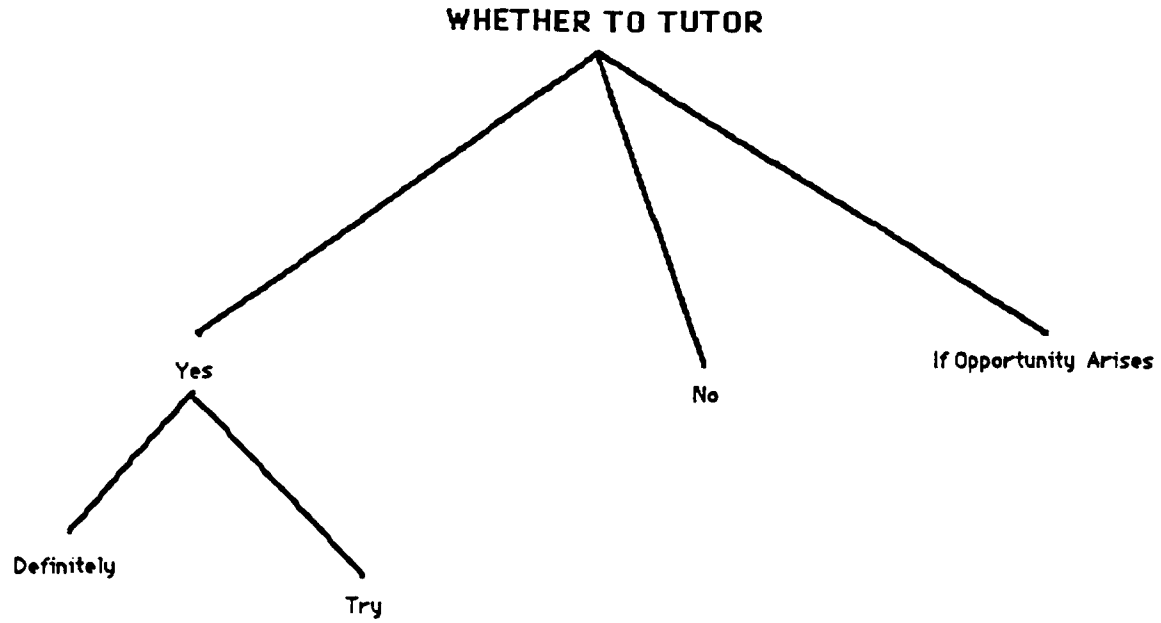


Figure 3.9: Whether to Tutor

2.1.1.3. Rules for YES-NO Decision

The rules that make Tutorial Planning Decision 2, the YES-NO decision, use tutorial knowledge types from the categories of Student, Domain, and Bugs. An example of a YES-NO rule is the following:

YES-NO-BUGS:DEFINITELY-RULE1

IF **Effects of BUG-X are important**
AND Effects of BUG-X are hard to understand
AND Cause of BUG-X is easy to understand
THEN Definitely Tutor BUG-X.

YES-NO-BUGS:DEFINITELY-RULE1 uses information about the causes and the effects of bugs. The IF clauses asks whether the effects of the bug are both 1) important and 2) hard to understand and whether the cause of the bug is easy to understand. The three IF clauses use two different kinds of features:

• Primitive Features Primitive features are part of the representation of students, the domain, and bugs. The first clause of YES-NO-BUGS:DEFINITELY-RULE1, which asks whether the effects of the bug are important, uses the primitive feature of the importance of the effect of the bug. All the features discussed in the first part of the chapter on the four categories of tutorial knowledge are primitive features.

• Computed Features Computed features are derived from two or more primitive features. The second and third clause of YES-NO-BUGS:DEFINITELY-RULE1 uses computed features. The difficulty of understanding the effect of a bug varies according to the ability of the student and the default difficulty of understanding the effect. For example, an effect is hard to understand if the student is not of high ability and the default difficulty of understanding the effect is high. A similar method is used to compute whether the cause of a bug is easy to understand.

Figure 3.10 shows how computed difficulty is derived from default difficulty and student ability. The entries in the cells of the table show the computed difficulty values that correspond to the joint values of student ability and default difficulty. This method of

computing difficulty values is used for the "difficulty" features listed in Figure 3.11. For example, the difficulty of tutoring a bug is a joint function of the student's ability and the default difficulty of tutoring the bug. Thus, the second and third clauses of the rule **YES-NO-BUGS:DEFINITELY-RULE1** use computed features.

A second example rule, **YES-NO-BUGS:DEFINITELY-RULE2**, is based on the features of the task specification, a tutorial knowledge type in Category 2, knowledge about the domain:

YES-NO-BUGS:DEFINITELY-RULE2

IF Task Specification in which BUG-X appears is important
AND the cause of the bug is not a slip
THEN Try To Tutor BUG-X.

YES-NO-BUGS:DEFINITELY-RULE2 insures that all bugs that are in important parts of the task specification e. g., the calculation of AVERAGERAIN; the output statements for AVERAGERAIN and MAXIMUM, are given a chance to be addressed.

Figure 3.12 gives an example rule for each of the four possible decisions that can be made about whether to address a bug in the tutoring plan. For example, in the third row is a rule that, if fired, results in a bug not being tutored. In this case, if a bug is hard to tutor and the importance of the bug is low, then it should not be tutored.

The previous discussion has focussed on the YES-NO decision for Singleton bugs. The method for making the YES-NO decision for a group of bugs is nearly the same. The decision method for a group

of bugs simply computes the maximum of the relevant features, such as importance or tutorial difficulty, for all the bugs in a group. The YES-NO decision is made according to the rules for the maximum values. If any bug, BUG 2, in a group is dependent on another bug, BUG 1, in the group e. g., one bug is a repair for another, then BUG 2 is addressed if BUG 1 is addressed.

In summary, the second Tutorial Planning Decision requires the tutor to decide whether or not to address each of the student's bugs. The tutor may decide definitely to address a bug or definitely not to address a bug. In addition, the tutor can take more of a wait-and-see attitude and decide to try to address a bug or see if an opportunity arises for the bug. Rules based on primitive features of students, bugs, and the domain, and features computed from the primitive features, are the basis of Tutorial Planning Decision 2.

	DEFAULT DIFFICULTY		
STUDENT ABILITY	HIGH	MEDIUM	LOW
HIGH	**MEDIUM**	**LOW**	**LOW**
MEDIUM	**HIGH**	**MEDIUM**	**LOW**
LOW	**HIGH**	**HIGH**	**MEDIUM**

Figure 3.10: Difficulty Computed from Ability and Default Difficulty

1. Difficulty of Understanding Cause of Bug
2. Difficulty of Understanding Effect of Bug
3. Difficulty of Tutoring a Bug

Figure 3.11: Difficulty Computed from Ability and Default

YES-NO DECISION	EXAMPLE RULES
DEFINITELY	IF Effects of BUG-X are important AND Effects of BUG-X are hard to understand AND Cause of BUG-X is easy to understand THEN Definitely Tutor BUG-X
TRY	IF Importance of BUG-X is High AND Student Knowledge about BUG-X weak THEN Try to Tutor
NO	IF Tutorial Difficulty of BUG-X is High AND Importance of BUG-X is Low THEN Do Not Tutor BUG-X
IF OPPORTUNITY ARISES	IF Importance of BUG-X is Medium AND Student Knowledge about BUG-X weak THEN Tutor if opportunity arises

Figure 3.12: Examples of Rules for Tutorial Planning Decision 2 -- YES-NO

3.2.1.2. Tutorial Planning Decision 3: ORDERING

The result of Tutorial Planning Decision 3 is that the groups of bugs, and the bugs within groups, are ordered in the tutorial plan. For example, if a tutor has no idea why a student made a bug then, if the bug is potentially important for the reasons discussed earlier in the chapter, then the tutor will focus on that bug first. For example, all 11 tutors in the Questionnaire Study said they would first address the four malformed variable updates in Figure 3.5, BUG 2, BUG 3, BUG 4, and BUG 5. The reason that they gave for addressing the malformed updates first was that they had no idea why the student made the bug but that, whatever the cause, it was serious and they wanted to find out before going further. The tutors therefore put all four bugs in a group of type Exploratory Diagnosis and planned to address that group first.

When more than one bug is in a group, the tutor must decide on the order in which to address the bugs. For example, when the tutor has made a grouping of bugs in which one is an incorrect repair for the other, the tutor first makes sure that the student understands the effects of the incorrect repair and then addresses the cause. This policy imposes an ordering on the bugs within the Repair grouping. The purpose of this section is to describe how groups of bugs are ordered in the tutorial plan and how bugs are ordered within groups of bugs. Each of these topics is treated in a separate subsection.

2.1.2.1. *Ordering Groups of Bugs*

Figure 3.13 shows the two prototypical orders of bug groups in tutorial plans. Whenever possible, tutors prefer to do diagnosis of important bugs before doing anything else. For example, as noted earlier, all tutors in the Questionnaire Study said that they would attempt to determine why the student made the malformed updates shown in Figure 3.5 before doing anything else. Although tutors prefer to engage in diagnosis of causes of important bugs first, there are circumstances which affect that ordering.

The chief reason that tutors do not perform diagnosis first is that the student is not especially good, or is unmotivated or dispirited, a common situation in tutoring novice programmers. The two branches shown in Figure 3.13 show the prototypical orders of groups of bugs for good students and for weak students.

• Good Students: When the tutor creates a tutorial plan for a good student, diagnosis of important bugs is the first issue addressed. As discussed earlier, a bug can be important either because of the part of the task specification it is in or because of the cause of the bug. For example, a bug caused by a misconception is more important that the same bug caused by a slip. Following the diagnosis of important bugs are Hard bug groups, as described in the section on Tutorial Planning Decision 1. Following the Hard bug groups are the remaining groups, ordered by default as shown in the box at the bottom of Figure 3.13. This topic is discussed below.

• Weak Students: When a tutor tries to help a weak or unmotivated student, the tutor is concerned with the student's

emotional state. Tutors try to avoid making students feel that they cannot find and fix bugs. Therefore, a tutorial plan for a weak student typically begins with an easy bug. As one tutor put it:

"This will give the student confidence that he can find bugs and make it more likely that he won't give up later when the going gets tough."

Following the motivational work with easy bugs are bugs for which it is important to perform diagnosis. Then, if the student is able, the tutorial plan addresses Hard bugs, followed by the order of the groups shown in the box at the bottom of Figure 3.13.

Once the tutor has addressed the motivational problems of weak students, performed any needed diagnosis, and addressed Hard bug groups, then the remaining bugs are considered. The remaining bugs are ordered according to the default order shown in the box at the bottom of Figure 3.13, subject to modification by the simple group ordering heuristics described below.

The default order for the remaining groups of bugs focuses first on making the program easy for the student to work with, then on important bugs and finally on programming techniques and skills that the tutor wants to teach the student. For example, the tutor might leave until last a group of bugs in which guards were missing from output statements. The tutor would not ordinarily care about fixing the output statements but might use the opportunity to teach a good student some skills of program testing, debugging, and so forth

The heuristics for altering the default order of the groups of remaining bugs are straightforward. Each of the heuristics expresses the intuition that "All other things being equal, prefer to

tutor bugs which meet the criteria exp essed by the heuristic". The foll wing descriptions of the heuristics identify the features which are the basis for altering the default order of groups. An example rule is given for each.

• Importance: All other things being equal, order of tutc ıg should begin with the most important bugs. The definition of an important bug is one which either 1) has effects on the program behavior that are hard to understand or hard to find, 2) is in an important goal of the task specification or 3) suggests the student has a misconception about an important programming concept. These three features are primitive features of, respectively, 1) bugs, 2) the task specification and 3) knowledge about programming. An example rule is

ORDER-GROUPS:IMPORTANCE1

I F Group 1 contains bugs or concepts that are more important than Group 2

THEN Address Group 1 before Group 2

• Knowledge Preconditions: Bugs or bug groups which provide knowledge that is required or useful for tutoring an other bug or bug group should be tutored before the bugs or bug groups that depend on them.

ORDER-GROUPS:KNOWLEDGE-PRECONDITIONS1

I F Group 1 contains bugs or concepts that are required for Group 2

THEN Address Group 1 before Group 2

• Program Behavior Preconditions: Bugs which make the program hard to interpret should be tutored as early as possible.

ORDER-GROUPS:PROGRAM-BEHAVIOR-PRECONDITIONS1

I F Group 1 contains bugs that make program behavior confusing for Group 2

THEN Address Group 1 before Group 2

• Difficulty of Tutoring: Bugs which are easy to tutor should be tutored before bugs that are hard to tutor, assuming that the rules for importance, knowledge preconditions, and program behavior preconditions have been satisfied.

ORDER-GROUPS:DIFFICULTY1

I F Group 1 contains bugs that are harder to tutor than Group 2

THEN Address Group 2 before Group 1

In summary, there is a default order for addressing bug groups. The default order depends on the student's ability and motivation. The default order is subject to alteration according to heuristics that are based on four, intuitive features such as importance and difficulty of tutoring.

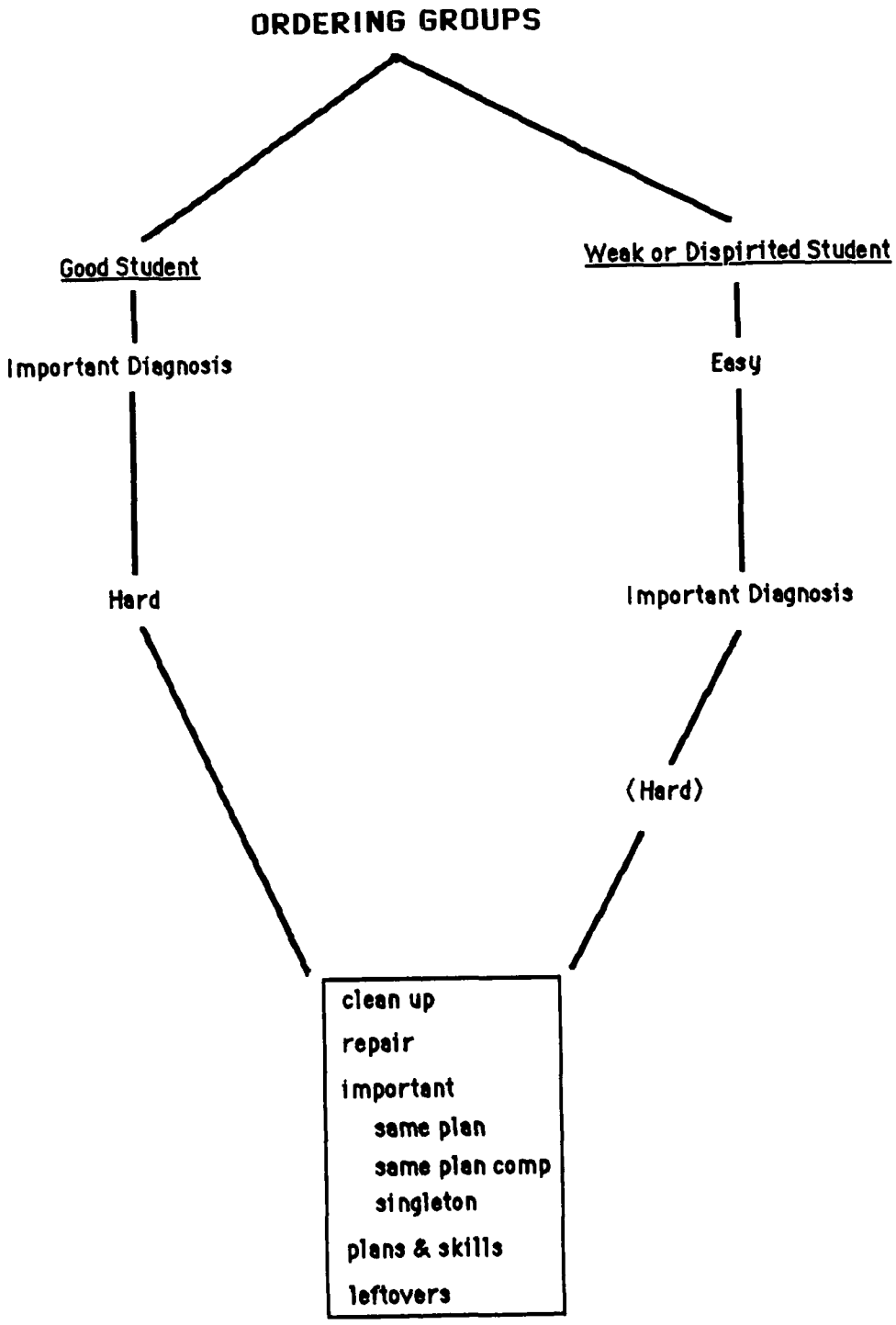


Figure 3.13: Ordering Group

2.1.2.2. Ordering Bugs Within Groups

Deciding the order of addressing bugs within groups depends upon two types of information. First, certain types of groupings dictate the order of bugs. Second, if the type of group does not dictate the order of addressing bugs within the group then heuristics, nearly the same as those which order groups of bugs, determine the ordering of bugs within groups. This section first describes constraints on the ordering of bugs within groups that are determined by the group and then describes the general heuristics for ordering bugs within groups.

Three types of groups impose an order on addressing bugs in the group. The two types of groups are Repair Groupings, Same Plan Groupings and Same Plan Component Groupings.

• Repair Grouping: In a Repair Grouping one or more bugs lead to the need for the repair (the cause bugs) and one or more bugs are the repair bugs. These two types of bugs are treated in a prototypical order. First the tutor identifies the cause of the repair. For example, in the case of the incorrect initialization of RAINDAYS to 1 rather than 0, shown in Figure 3.2, the tutor would focus the student's attention on the fact that the calculation of AVERAGERAIN could result in a runtime division-by-zero-error if RAINDAYS were initialized to 0. Second, the tutor would focus on the repair bug, in this case the initialization of RAINDAYS to 1. Third, the tutor would help the student to identify unwanted side effects of the repair. In this case the unwanted side effect is that the value of RAINDAYS is incorrect when the first value entered by

the user is the sentinel. Finally, the tutor would help the student choose or generate an alternative plan for handling the cause, in this case the potential division-by-zero error.

• Same Plan Grouping: The order of bugs in a Same Plan Grouping is determined by the focus of the plan component in which each bug occurs. The focus of each plan component is a feature that is stored with the plan component. Thus, the update of a variable is more focal than the initialization, which in turn is more focal than the output guard. For example, if the student neglected to include the update for MAXIMUM, the initialization of MAXIMUM, and the output guard for MAXIMUM, then the tutor would first address the missing update, then the missing initialization, and finally the missing output guard.

• Same Plan Component Grouping: The order of bugs in a Same Plan Grouping is determined by the importance of the bugs in the group and the degree to which the bug illustrates the plan component. For example, if a Same Plan Component Grouping contains several missing initialization bugs, then the bugs will be addressed in order of importance. The importance of the variable is determined by whether it is a housekeeping variable, such as RAINDAYS or a key variable, such as TOTALRAIN or AVERAGERAIN.

The heuristics that order bugs within groups are essentially the same as those which order groups of bugs. As in the case of the heuristics that order groups of bugs, the heuristics that order bugs within groups depend upon importance, knowledge preconditions, program behavior preconditions, and difficulty of tutoring. The rules

that express these heuristics are applied to each group when that group is the one being addressed by the tutor.

In summary, the problem of ordering bugs for tutoring has two aspects. First, groups of bugs are ordered in a default sequence that can be modified depending upon the ability of the student and features of the bugs in the group such as importance, difficulty of tutoring, and so forth.

Second, bugs within groups are ordered in two ways. First, some bug groupings, such as repair groupings, have a built-in order for addressing the bugs. Second, heuristics for ordering bugs within groups are used when the bug grouping does not dictate an order. The heuristics for ordering bugs within groups rely on the same features as the heuristics that order groups of bugs.

3.2.1.3. Tutorial Planning Decision 4: OBJECTIVES

Making Tutorial Planning Decision 4 results in the assignment of tutorial objectives to each group of bugs in the tutorial plan. For example, if in the attempt to repair one bug a student makes a second bug, then the tutor's objectives would depend on whether the repair is good or bad. If the repair fixes the initial bug, as in the program shown in Figure 3.3, the tutorial objective would be to teach the student a plan that avoids the need for repair. If the repair is incorrect, as in the program shown in Figure 3.2, then the tutor would have two objectives. The first objective would be to lead the student to see that the repair is incorrect. The second objective would be to teach the student the preferred plan, just as

in the case of the correct repair. The purpose of this section is to describe 1) the conceptual knowledge about objectives that tutors use when they construct tutorial plans and 2) the rules that assign objectives to tutorial plans.

2.1.3.1. Knowledge of Objectives

Figure 3.14 shows the possible objectives that can b assigned to a tutorial plan. As the first level of the tree in the figure shows, there are four main types of objectives, each of which has subtypes:

• Teaching Objectives: Teaching objectives are intended to provide the student with knowledge or skills that the student does not have or has not mastered, or need correction. For example, the student who made BUG 5, the misplaced update of TOTALRAIN below the loop in the program shown in Figure 1.1, is confused about the concept of the scope of a loop. Thus, the tutor who wants to help this student would establish a teaching objective to make sure the student understands the concept of scoping of loops.

BUG 1 also suggests that the student may not have good skills in testing programs because an effective simulation would have detected the fact that the assignment of 0 to RAINFALL destroys the initial value of RAINFALL acquired by the input statement. The tutor might, therefore, establish a teaching objective to improve the student's skill in hand simulating programs.

The two remaining types of teaching objectives are to teach plans and to eliminate misconceptions. The first example in this section, of the tutor deciding to teach the student a way to solve the

task specification that avoids the need for a repair, was an objective to teach plans. Sometimes students have active misconceptions that require correction. If the tutor discovered that the student had an active misconception about the scope of a loop, e. g., that the scope of a loop is any consecutive sequence of BEGIN-END blocks, then the tutor would set the objective to eliminate that misconception.

• Fix Program Objectives: Tutors frequently establish objectives to help students fix their programs. Sometimes the code is the focus of the tutoring, as in the case of the spurious initialization of RAINFALL, shown as BUG 1 in the program in Figure 1.1. In this case the tutor would probably ask the student to hand simulate the buggy lines of code with the intent of leading the student to see how the spurious line destroys the initial, needed value of RAINFALL acquired with the input statement.

An objective to fix the program does not mean that the code is the focus of tutoring. A tutor can also decide to help the student fix the program but focus on the student's *plans* for achieving a goal rather than on the code. For example, if the student calculates AVERAGERAIN in the loop then the tutor may decide to help the student to choose a preferred plan, namely calculating AVERAGERAIN after the termination of the loop. The tutor's objective is to help the student fix the program and in this case the tutor focuses on the student's decision to calculate AVERAGERAIN in the loop. I. e., the tutor will focus on the student's plan for calculating AVERAGERAIN.

Finally, if the student has solved the wrong problem, for example, calculating AVERAGERAIN using RAINDAYS rather than

TOTALDAYS as the divisor, the tutor may help the student to fix the program but will focus on the student's incorrect *goal*, namely failing to calculate AVERAGERAIN over the whole period during which the user entered rainfall data.

• Session Management Objectives: Session Management objectives are typically in service of other objectives. Tutors establish session management objectives when they want to control the way in which the tutoring session proceeds. For example, a tutor who wanted to run the program shown in 1.1 and ask the student questions about its behavior would know that the infinite loop caused by the missing input statement for RAINFALL would make the behavior of the program hard for the student to understand. One of the first priorities of the tutoring session would thus be eliminating BUG 4, the missing input statement for RAINFALL that causes the infinite loop. The reason for working on BUG 4 would be to clean up the behavior of the loop before working on the misplaced update of TOTALRAIN. Thus, the tutor's objective in working on BUG 4 would be a session management objective about program behavior preconditions.

If a tutor decides that a student needs to understand, or be made aware of, some facts or concepts before working on a particular problem, then the tutor would establish a Knowledge Precondition objective. For example, a tutor may decide that a student should understand how interactive loops work in general before explaining that there are two main kinds of interactive WHILE loops, one in which the input statement is at the beginning of the

loop and one in which the input statement is at the bottom of the loop.

In addition, the tutor may decide that a student should understand the serious effects of a bug before teaching the student a plan that avoids the bug. In this case the tutor would establish a Knowledge Precondition objective to make the student aware of the effects of the bug.

Finally, tutors understand that poor students can become discouraged when they solve difficult problems. Tutors may begin a tutoring session with a bug that will be easy to fix and inspire the student with confidence from the success before proceeding to more difficult bugs. Equally, tutors know that good students often get bored when the work becomes too easy; to prevent boredom tutors may have the session management objective of sparking the student's interest and so begin the tutoring session with a difficult bug.

• Diagnostic Objectives: As Figure 3.14 shows, diagnostic objectives are either exploratory or confirmatory. A tutor who has no idea why a student made one, or several similar, bugs would establish the objective to explore causes. For example, the four malformed update bugs in Figure 3.5 (BUG 2, BUG 3, BUG 4, and BUG 5) would lead a tutor to explore why the student made the bugs. On the other hand, if the tutor had a strong hypothesis about why a student made a bug -- such as the misconception that the scope of a WHILE loop includes any sequence of BEGIN/END blocks -- then the tutor would establish a diagnostic objective to confirm that this misconception is the cause.

In sum, there are four main types of objectives which tutors can establish as part of a tutorial plan. First, teaching objectives are intended to provide the student with some kind of knowledge or to correct existing beliefs. Second, tutors often help students fix their programs. In addition to focusing on the code, this frequently means fixing the program from the perspective of the plans the student has used or the goals the student has attempted to solve. Third, session management objectives control the order of topics and the interest of the student in the tutoring session. Finally, diagnostic objectives are established when the tutor wants to determine the cause of some action of the student.

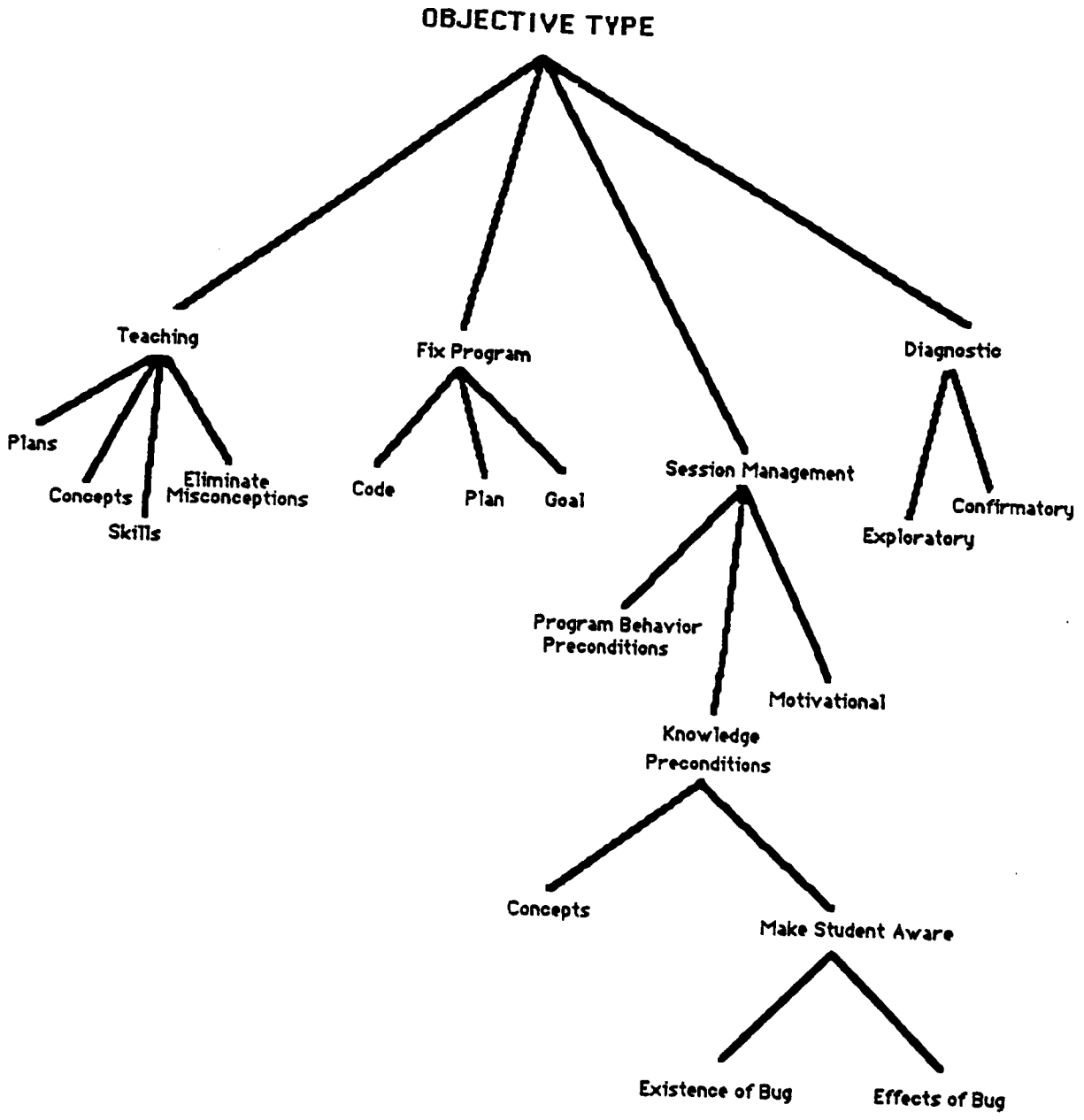


Figure 3.14: Objective Types

2.1.3.2. Objective Selection Rules

Tutorial objectives are established in two ways. One set of rules for establishing tutorial objectives depends upon the type of group to which the objective is being assigned. A second set of rules assigns session management objectives to any type of groups.

2.1.3.2.1. Objective Selection Rules Based On Group Type

Figure 3.1 displays the types of groupings that tutors place bugs into. Each of the groupings requires different objectives. Objectives that depend upon the type of grouping are organized in packets that handle special cases. For example, the following rule assigns an objective to a bug group in which the student has made a correct repair to an initial bug:

OBJECTIVES:GOOD-REPAIR1

IF BUGGROUPTYPE is repair
AND the repair is good
THEN Assign the objectives
Teach Preferred Plan

OBJECTIVES:GOOD-REPAIR1 means that if the student has corrected one bug with an inelegant patch, then the tutor should try to teach the student a better way to achieve the task specification goal that led to the initial bug. The next rule handles the case in

which the student has repaired an initial bug but the repair does not work:

OBJECTIVES:BAD-REPAIR1

IF BUGGROUPTYPE is repair

AND the repair is bad

THEN Assign the objectives

Make Student Aware of Repair Failure

Teach Preferred Plan

This rule assigns two objectives to a group in which the student has made an unsuccessful repair to an initial bug. First, the tutor should make sure that the student is aware of the fact that the repair is not good. The assumption is that the student did not know that the repair was incorrect and needs to see that it is, and why. Once the student knows that the repair is bad, then the tutor should try to teach a new plan that avoids the bug that led to the need for the repair. Thus, **OBJECTIVES:BAD-REPAIR1** is like **OBJECTIVES:GOOD-REPAIR1** with the exception that the tutor first makes sure that the student knows that the attempted repair does not fix the original bug.

Figure 3.15 shows examples of rules that assign objectives to groups according to the type of group. Each of the example rules is one of several rules that assign objectives for groups of that type. As the example rules show, the rules depend upon the group type, as well as upon features of the bugs in the group. For example, the third rule, OBJECTIVES:SINGLETON1, which is one of the rules that

assigns objectives to Singleton groups, uses the type of cause of the bug to assign tutorial objectives. The rule shown in the figure handles the case in which the cause of the bug is a misconception. This cause leads to a teaching objective to eliminate the misconception, one of the types of objectives shown in Figure 3.14.

GROUP TYPE	EXAMPLE OBJECTIVE ASSIGNMENT RULES
GOOD REPAIR	OBJECTIVES:GOOD-REPAIR1 IF BUGGROUPTYPE is Repair AND the repair is good THEN Assign objectives 1. Teach Preferred Plan
SAME PLAN COMPONENT	OBJECTIVES:SAME-PLAN-COMPONENT1 IF BUGGROUPTYPE is Same Plan Component AND the plan component is housekeeping AND the group is Definitely Tutor THEN Assign objectives 1. Identify one bug 2. Identify Preferred Plan for bug 3. Make Student Aware of remaining bugs
SINGLETON	OBJECTIVES:SINGLETON1 IF BUGGROUPTYPE is Singleton AND the cause of the bug is misconception THEN Assign objectives 1. eliminate misconception

Figure 3.15: Examples of Rules that Assign Objectives to Tutorial Plan

2.1.3.2.1. Objective Selection Rules Not Based On Group Type

In addition to rules based on the type of group, there are rules for assigning tutorial objectives that are based on other features. Aside from group type, the main reason for assigning tutorial objectives is to provide the student with knowledge preconditions -- one type of Session Management objective -- as shown in Figure 3.14.

For example, in the case of the student who incorrectly initialized the counter TOTALDAYS to 1 to avoid division by zero in the calculation of AVERAGERAIN, as shown in Figure 3.2, the tutor may establish the objective of ensuring that the student understands the concept of initialization, and the effects of initialization, before helping the student with the preferred plan for guarding the calculation of AVERAGERAIN.

Tutorial objectives can also be established to maintain a student's motivation. For example, if the student's ability is low, and the student has made several bugs, then the tutor may decide to begin the tutoring session with an easy problem to stimulate the student's interest. This problem is discussed in the section on making Tutorial Planning Decision 4 -- when in the tutorial plan to address bugs.

In summary, rules that assign tutorial objectives to the tutorial plan are of two types. The first, major type of rule assigns objectives to groups according to the type of group. For example, the objectives assigned to a Repair group of bugs is different from the objectives assigned to a group of bugs which share a common

plan component. Additional objective selection rules are based on features of the tutoring situation beyond the type of group. These rules assign objectives that are primarily directed toward managing the tutorial session.

3.2.1.4. Tutorial Planning Decision 5: STRATEGY

Tutorial Planning Decision 5 requires the tutor to select strategies to achieve tutorial objectives. For example, a tutor might have the objective of making sure that the student is aware of the effects of a particular bug, say a missing guard against division by zero in the calculation of AVERAGERAIN; this bug is shown as BUG 6 in Figure 1.1. If the objective is to make the student aware of the effects of the missing guard against division by zero, then the tutor might employ the strategy of asking the student to hand simulate the part of the program after the loop, where the calculation of AVERAGERAIN is performed. The tutor would ask the student to hand simulate the program segment in the case where the user does not enter any data. This strategy will cause the student to attempt the division by zero and will make it possible to lead the student to see the effect of the missing guard, namely a runtime error caused by the attempt to divide by zero.

The purpose of this section is to describe the tutorial strategies that tutors know about and to show rules that govern the selection of strategies to achieve tutorial objectives. The section is divided in two subsections. The first subsection describes the

strategies that tutors use. The second section describes the rules that are used to select strategies.

2.1.4.1. Knowledge of Tutorial Strategies

Figure 3.16 shows the four main strategies that tutors use to achieve their objectives when they assist programming students. Each of the strategies can be performed in one or more situations, which are designated by the branches emerging from each of the strategies. For example, a tutor might ask a student to justify either a plan for a goal that the student has stated, e. g., the goal of calculating a total for TOTALRAINFALL, or to justify a goal for a particular task specification, e. g., to find the average amount of rain that fell in a designated period. The purpose of this section is to describe the four main strategies that tutors use to help programming students. Each of the four strategies is described and examples are given of the most important cases.

• Justify: Tutors ask students to justify their actions when they want the student to understand why an action is correct, or to discover why an action is incorrect. For example, a student might have added one to TOTALRAINFALL on each iteration of the loop, rather than adding in each new value of DAILYRAIN. In this case, the tutor might ask the student to justify the use of the "add 1" plan for the running total with the expectation that, as the student attempts to formulate the justification, the bug will be detected. This case of the justification strategy is indicated by the leftmost leaf in Figure 3.16.

The second reason for using the justify strategy arises when the student has misinterpreted a task specification, defined an incorrect goal to achieve, and used the appropriate plan for the incorrect goal. For example, some students misinterpret the Rainfall Assignment to mean that AVERAGERAIN should be the average amount of rain that fell on days when it *rained*. The student then divides TOTALRAIN by RAINDAYS to calculate AVERAGERAIN. Thus, the student's goal is wrong but the plan for the student's goal is appropriate. In this situation a tutor might ask the student to justify the calculation of the average rainfall for rainy days. The tutor expects that the student will detect the error as the justification for the goal is formulated.

• Explain: Tutors ask students to explain some aspect of the program when they want the student 1) to articulate what the program does and 2) to draw some conclusions from the description e. g., that the program does the wrong action. In contrast to the strategy of justification, which focuses c the reasons for an action, explanation focuses on the actions.

For example, if a tutor wanted to make the student aware of the fact that the wrong variable is output in BUG 7 in Figure 1.1, the tutor might ask the student to explain "what that line of code is doing". The student would probably reply with the statement that the code is printing out the number of rainy days. The tutor would then ask the student to explain how that number is written out. The student at this point would probably detect the fact that the value of the wrong variable was being printed out. If the tutor used the strategy of justification, the tutor would probably ask the student

why the student printed out the value of RAINFALL in the output statement with the expectation that the student would immediately see that RAINFALL is the wrong variable.

• Simulate: Tutors ask students to simulate their programs, or segments of their programs, when they want the student to obtain information about the behavior of the program. There are two types of simulation, indicated by the two branches stemming from Simulate in Figure 3.16.

First, tutors ask students to hand simulate programs with particular values for certain variables. For example, the tutor might ask the student to simulate a Rainfall Program when the only value the user enters is 99999, the sentinel. In this case the tutor would want the student to detect the division by zero error at the calculation of AVERAGERAIN because TOTALDAYS would still be 0, its initialization value.

The second kind of simulation does not depend upon specific values for variables. The tutor asks the student to simulate the flow of actions in the program, labeled as Simulation of Goals in Figure 3.16. For example, the tutor might want the student to detect the misplaced update of a variable, such as BUG 5 in Figure 1.1. The tutor would ask the student to describe the sequence of actions that occurs in, and after, the WHILE loop of the program. The tutor does not care about any particular values of variables. Rather, the tutor wants to help the student to understand the order in which events occur i. e., goals are satisfied and how this affects the update of the TOTALRAIN.

• Simplify: Tutors asks a student to work on a simplified form of a problem when the student is confused or the tutor is trying to determine why the student made a bug. As Figure 3.16 shows, there are three types of simplification: of concepts; of plans; and of goals. Each of these three types of simplifications can be used for two reasons.

First, if the student is confused by an issue and the student's program is too complex to illustrate the issue, then the tutor might use a simplification. For example, if the student is confused about the concept of the scope of the WHILE loop, which is needed for the Rainfall Assignment, the tutor may construct a simplified WHILE loop, perhaps with only an input and an output statement. Using this simplified example, the tutor might progressively add statements to the loop until it satisfied the Rainfall Task Specification. At each step the tutor would make sure that the student understood the effects of the new additions.

Second, tutors simplify when they want to determine why a student made a bug. For example, the student who made the malformed updates shown in Figure 3.5 is confused about several issues which are not apparent from the malformed code. Tutors in the Questionnaire Study said that they would try to find out what the student knows about assignment, updating, and running totals by using a simple version of them, asking the student questions about the behavior of the code, and progressively adding complexity until the cause of the bug was identified.

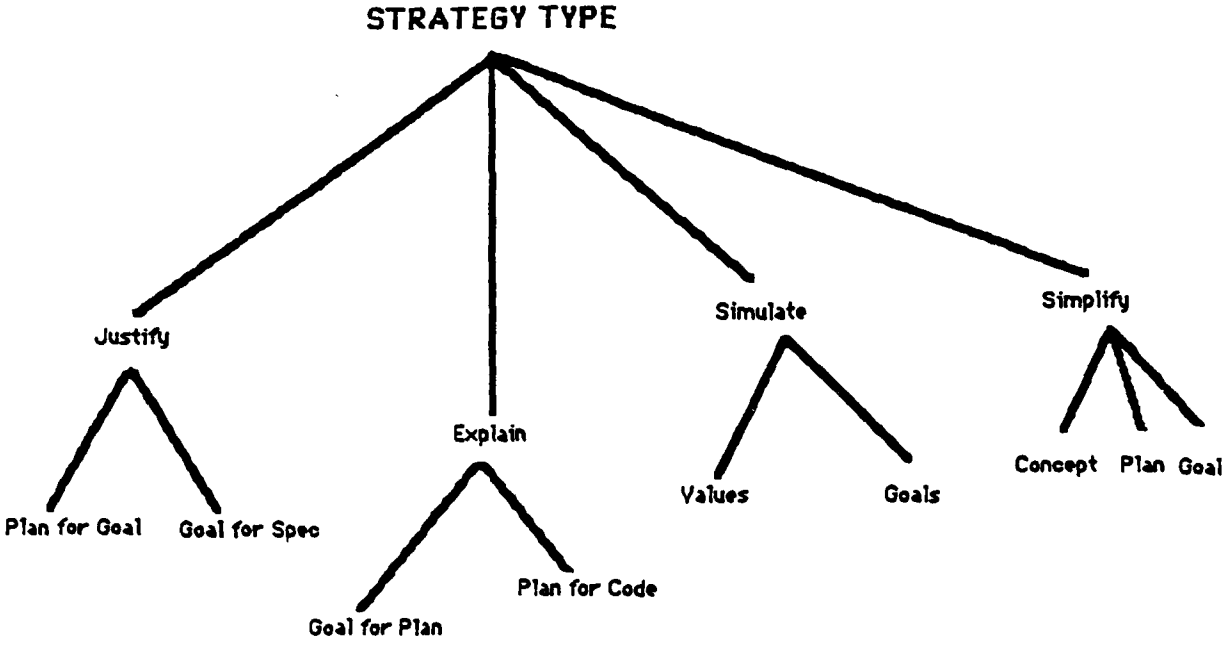


Figure 3.16 Strategy Type

2.1.4.2. Rules that Select Tutorial Strategies

The foregoing discussion of the four strategies presented examples in which the tutor requested the student to justify, explain, simulate, or simplify. In addition to *requesting* the student to perform an action, such as simulating a section of code, tutors also *provide* justifications, explanations, simulations, or solutions to simplified problems.

For example, the tutor may want the student to draw conclusions from the behavior of the processing loop in a buggy Rainfall Program. However, because the student is weak, the tutor does not believe that the student can 1) perform a simulation of the

loop, 2) observe the result of the simulation, and 3) draw conclusions from the simulation. In this case, the tutor might perform the simulation for the student and even direct the student's attention to relevant aspects of the simulation with the goal of supporting the student in drawing the desired conclusions e. g., that the update of TOTALRAIN occurs only once, after the main loop is finished, as is the case for BUG 5 in 1.1.

2.1.4.2.1. Selecting Tutorial Strategies

The rules that select a tutorial strategy must make two decisions. First, the rules must select the appropriate tutorial strategy i. e., justify, explain, simulate, or simplify. Second, the rules must determine how much assistance to provide the student i. e., should the tutor *ask the student* to perform a hand simulation of a segment of code or should the tutor *provide* the simulation for the student? The purpose of this section is to describe how the two decisions are made.

Figure 3.17 shows how the selection of the four tutorial strategies depends upon the objective that the tutor wants to achieve. The following examples describe situations in which each tutorial strategy is appropriate to the tutor's objective and states one of the rules that is used to select the strategy in that situation. Each objective has more than one associated rule that covers the various cases that arise for that objective. The rule shown here for each objective is intended to be a straightforward example.

• Objective -- Make Student Aware of Effects of Bug: If the tutor's objective is to make the student aware of the effects of a bug, such as the spurious initialization of RAINFALL shown as BUG 1 in 1.1, then the tutor would focus the student's attention on the *behavior* of the program and select the strategy of simulation. A rule for selecting the strategy for the objective of making the student aware of the effects of a bug is:

STRATEGIES:MAKE-STUDENT-AWARE-EFFECTS1

IF Objective is Make Student Aware of Effects of Bug

THEN Assign the strategies

simulation

This rule simply says that if the objective is to make the student aware of the effects of a bug, then use the strategy of hand simulation.

• Objective -- Teach Student a Plan: If the tutor has the objective of teaching the student a plan for a goal, such as using an interactive loop to calculate a running total, the tutor would want the student to focus on the relationship between a goal (calculate a running total) and plans for doing so (using an interactive loop). This would lead the tutor to choose the strategy of explaining the connection beginning, perhaps, with simplifications of the plan. A rule for selecting the strategy for the objective of teaching the student a plan is:

STRATEGIES:TEACH-PLAN1

IF Objective is Teach Student Plan for Goal

THEN Assign the strategies

explain how subgoals correspond to plan components

OPTIONAL: use simplifications

This rule says that if the objective is to teach the student a plan for a goal, the the focus should be on explaining the way in which the goal's subgoals are implemented by the components of the plan. For example, the goal to calculate a running total has several subgoals which are implemented by components of the Running Total Plan. That is, the plan component that performs the addition corresponds to the "add in consecutive values of input variable to summation variable" subgoal; the plan component that inputs a new value corresponds to the "get a new value" subgoal; the plan component which checks for a termination condition corresponds to the "stop when input ended" subgoal. The second clause of the THEN statement in the rule says that it may be necessary or useful to use simplifications of both the goals and the plans in addition to explaining the how the full plan implements the entire goal.

• Objective -- Eliminate Misconception: If the tutor's objective is to disentangle a student's confusions about doing updates of variables, the tutor might "begin at the beginning" and ask for an explanation of the student's beliefs about updates. Then the tutor might ask the student to solve a progression of update problems ranging from very simple to more complex. One of the

rules for selecting the strategy for the objective of eliminating misconceptions is:

STRATEGIES:ELIMINATE-MISCONCEPTION1

IF Objective is Eliminate Misconception

THEN Assign the strategies

use simplified counterexample

explain incorrect concept

explain correct concept

This rule says that if the objective is to help a student to eliminate a misconception, then the tutor should generate a counterexample that shows the problem with the student's beliefs, then explain the incorrect concept, and finally, to explain the correct concept. As will be seen in the discussion of the second decision required for strategy selection, namely how much help to provide the student, assigning the strategy "explain" can result either in the tutor providing an explanation of the incorrect concept (clause 2 of the THEN statement in the rule) or asking for an explanation of the belief from the student.

• Objective -- Fix Program Plan: If the tutor has the objective of helping the student to fix a program plan, such as calculating AVERAGERAIN in the correct location after the loop rather than inside the loop, the tutor's objective will be to focus on the connection between the location feature of the student's plan and the location feature of the task specification goals. The tutor will therefore select the strategy of justification of the plan

location, perhaps with an explanation of why one location is preferred over another. For example, the tutor might ask the student to justify the necessity of calculating AVERAGERAIN in the loop and then might provide an explanation of why placing the calculation in the loop is not necessary. A rule for selecting the strategy for the objective of fixing a plan in the student's program is:

STRATEGIES:FIX-PROGRAM-PLAN1

IF Objective is Fix Program Plan

THEN Assign the strategies

justify the current plan

OPTIONAL: explain why unpreferred

justify preferred plan

This rule says that if the objective is to help the student fix a plan in the student's program, then the tutor should focus on the connection between the plan and the goals it satisfies by using a strategy of justifying the plan. Next, the tutor may work on an explanation of why the current plan is not the one of choice. Finally, the preferred plan is justified, either by the student or by the tutor.

• Objective -- Diagnose Reason for Bug: Finally, the tutor may have the objective of diagnosing the reason that a student made one or more bugs. For example, if the tutor wants to discover why the student made the four malformed update bugs shown in Figure 3.5, the tutor would want to focus on the student's beliefs about updating variables. This leads to selection of the strategy of asking for an explanation of the student's beliefs and perhaps asking the

student to solve some simplified updating problems. A rule for selecting the strategy for the objective of diagnosing the reason a student made one or more bugs is:

STRATEGIES:DIAGNOSE-CAUSE1

IF Objective is Diagnose Cause of Bug

THEN Assign the strategies

ask student to explain

the goal the student was attempting to solve

the reason for plan in which bug is found

the reason for code in which bug is found

OPTIONAL: use simplification

OPTIONAL: use justification

This rule says that if the objective is to determine why a student made a bug then (the student will) explain the reason for writing the undiagnosed code, selecting the wrong plan, and trying to achieve the incorrect goal; Two clauses specify optional strategies that may help to achieve a diagnosis. For example, sometimes asking the student to solve a simpler case of the goal that led to the bug can provide useful diagnostic information.

TUTORIAL OBJECTIVE	FOCUS	STRATEGY
Make Student Aware of Effects of Bug	behavior of program	simulation
Teach Plan	connection between goal and plans	explain (+ simplify)
Eliminate Misconception	student's beliefs	explain (+ simplify)
Fix Program Plan	connection between plan and goals	justify (+ explain)
Diagnostic	student's beliefs	explain (+ simplify) justify

Figure 3.17: Basis for Selecting Strategies

2.1.4.2.2. Deciding How Much Help to Give

The decision about how much help to give the student is a straightforward function of 1) the importance of what is being tutored (the object of tutoring e. g., the scope of a WHILE loop) and 2) the difficulty of tutoring what is being tutored, which was described in the section on Tutorial Planning Decision 2 and is presented in Figure COMPUTFEAT1.[1]

[1]The general problem of combining features to derive a numerical value is complex. The issue was resolved in the dissertation by using a simple linear sum of the component features. This scheme is sufficient to make tutorial planning decisions that result in reasonable tutorial plans, as discussed in Chapter 6.

The importance of the object of tutoring is computed from the importance of 1) the task specification, 2) programming knowledge involved, 3) causes of bugs and 4) effects of bugs. Each of these kinds of knowledge has a primitive feature that defines its importance. For example, the calculation of AVERAGERAIN, one goal in the task specification for the Rainfall Assignment, is more important than the goal of initializing the counter for RAINDAYS; knowledge of iteration and how to use a WHILE loop to make an interactive program is more important than knowledge of runtime errors that result from division by zero; a bug which is caused by a misconception is more important than the same bug caused by a slip; a bug which has a pervasive effect, such as infinite looping, is more important than a bug which has minor effects, such as the counter for TOTALDAYS being off by one. The calculation of the importance of the object of tutoring is a simple sum of the importance of the four components.

There are two possible values for the amount of help to give to the student, namely *Request* and *Give*. (Perkins and Martin, 1986 discuss several issues bearing on how much help to give students.) For example, if a tutor wants a good student to reason about the results of a simulation of the main loop in a Rainfall Program, the tutor would ask the student 1) to perform the simulation and 2) then describe "what is going on?". If the tutor wants a weak student to reason about the results of a simulation of the main loop, the tutor would perform the hand simulation for the student, making sure that the student knows what values the variables take on, and would

then ask the student "what is going on?" to lead the student to draw conclusions from the result of the simulation.

The intuition behind the decision about how much help to give the student is expressed in the following general rule:

STRATEGIES:SELECT-HELP-LEVEL

IF Student is weak on strategy

AND the strategy is not the object of tutoring

THEN Assign help level GIVE to strategy

This rule says that students should not be made to do actions which are both 1) not the focus of tutoring and 2) likely to interfere with helping the student to understand the focus of tutoring.

For example, suppose a student has placed the calculation of AVERAGERAIN in the main loop. This is unnecessary because the calculation of AVERAGERAIN needs to be done only once after the termination of the main loop. The tutor's objective is for the student to select an alternative plan for the calculation of AVERAGERAIN, namely placing the calculation of AVERAGERAIN after the main loop. The tutor's approach is to have the student evaluate a justification of placing the calculation *in* the loop with the intent that the student will see that placing the calculation in the loop is redundant. If the student is weak, the tutor might provide a false justification of the placement of the calculation in the loop by saying that AVERAGERAIN must be calculated on each iteration of the loop. The student should recognize that the tutor's statement is false and then select the plan of placing the calculation of

AVERAGERAIN after the main loop. If the student were strong, the tutor might ask the student both 1) to justify placing the calculation of AVERAGERAIN in the loop and 2) to select an alternative plan for the calculation.

It should be noted that a strategy itself can be the focus of tutoring. For example, tutors frequently teach students how to hand simulate their programs, a skill that is important for program testing and debugging. If the strategy itself is the object of tutoring then the foregoing description of whether tutors request the student to perform a strategy or give the student the "answer" depends upon how important the strategy is to teach (they all are important) and how hard it is to teach (simplification is especially hard to teach). The foregoing analysis of how the decision is made applies to strategies when they are objects of tutoring.

In summary, this section has described four tutorial strategies that are widely used by tutors. The four strategies described in this section are justification, explanation, simulation, and simplification. Tutorial strategies are selected according to the tutor's objectives. In addition to deciding which strategy to use to attain a tutorial objective the tutor must also decide how much help to give the student with the strategy. The general rule for deciding how much help to give the student with the strategy holds that the performance of the strategy e. g., hand simulation should not interfere with the main objective of the tutoring e. g., teaching the student about the scope of a WHILE loop.

3.2.1.5. Summary

The five Tutorial Planning Decisions comprise the tutor's knowledge about tutorial planning. Making each of the tutorial decisions results in further specification of the tutorial plan. The tutorial plan specifies

- which bugs should be addressed together: Tutorial Decision 1.
- which bugs to tutor: Tutorial Planning Decision 2.
- what order to tutor bugs in: Tutorial Planning Decision 3.
- what to accomplish in tutoring the bugs: Tutorial Planning Decision 4.
- how to tutor each bug: Tutorial Planning Decision 5.

The five Tutorial Planning Decisions depend upon three categories of knowledge, namely knowledge about students, knowledge about the domain, and knowledge about bugs. Each of the foregoing categories has several types of knowledge within it. For example, knowledge of causes of bugs, locations of bugs, and effects of bugs are all in the category of knowledge about bugs. These three categories of knowledge were defined in the previous chapter.

3.3. Chapter Summary

This chapter has described tutorial knowledge about teaching, the fourth of the categories of knowledge identified in Figure 2.3. The main focus of this chapter was on the Five Tutorial Planning Decisions and how the knowledge described in the previous chapter is employed to make them.

The next chapter gives an overview of the generation of tutorial plans. The overview of tutorial plan generation makes the point that there can be many different plans for the same buggy computer program and that the plan that is generated depends upon 1) the tutorial planner's knowledge and 2) the tutorial planner's control structure. The subsequent chapter describes TP, the computer program that uses the knowledge described in this and the previous chapters to generate tutorial plans for multi-bug novice programs.

Chapter 4

TUTORIAL PLANS AND TUTORIAL PLANNERS

4.1. Introduction

Previous chapters have introduced the issue of multiple bugs, identified the five Tutorial Planning Decisions that confront a tutorial planner that must cope with multiple bugs, and described the main types of knowledge that are brought to bear on the five Tutorial Planning Decisions. Along the way, fragments of tutorial plans were described and some rules that might guide tutorial planning were suggested. There is, of course, more than one possible tutorial plan for any multi-bug program. Some plans are very simple, some are complex. Generating very simple tutorial plans does not require much complexity of knowledge or control structure. On the other hand, more complex tutorial plans, which would help students more than a simple plan, require much more knowledge and more complex control structures to generate.

The purpose of this chapter is to give an idea of different kinds of plans that tutorial planners might generate and to display the knowledge and control structures that would be necessary to generate them. This chapter illustrates, for the same set of multiple bugs, tutorial plans that vary in complexity. The discussion presents a progression of four tutorial plans for the set of bugs. The

progresssion begins with a very simple plan and ends with one that is much like the plan that human tutors generate. After the four tutorial plans have been described, the control structures and knowledge required to generate each of them are discussed. Finally, the relationship between variability and tutorial planners is briefly discussed.

The main purpose of this discussion is to prepare the reader for the next chapter which describes TP, the tutorial planner that is the focus of the dissertation. As the reader might anticipate, the fourth, most complex, tutorial planner described in this chapter is similar to TP.

	which to tutor	ordering	grouping	strategy
Plan1	ALL	LINEAR	NONE	CANNED
Plan2	important	LINEAR	NONE	CANNED
Plan3	important	clean up first	NONE	CANNED
Plan4	important first	clean up then lead in	lead in to import. bug	based on stud. ability

Figure 4.1: Attributes of Plans and Four Example Plans

4.2. Tutorial Plans

The following discussion uses the program shown in Figure 3.6 to illustrate different plans for the same bugs. Although the student who wrote the program made nine bugs, only six are considered in an effort to keep the following discussion as simple as possible. The discussion builds up from a simple tutorial plan that an experienced tutor would probably not use to a plan that a human tutor might use to help the student who wrote the program. The first plan is very much like one that the original PROUST (Johnson, 1985) would use. The second plan is slightly different in that it makes some decisions about which bugs to tell the student to work on. The second plan is similar to the one that PROUST, enhanced by a few very simple rules for managing multiple bugs, might use (See Appendix I). The third and fourth plans are more complex than any so far used with PROUST, or any similar tutoring system.

The differences between the four tutorial plans is shown in Figure 4.1. The columns represent the attributes of the tutorial plans that we have been considering. The rows indicate the values of the attributes for each of the plans Entries in capital letters mean that the tutorial planner does not reason about the attribute at the head of the column and that decisions about the attribute are fixed by the program's control structure. Entries in lower case mean that the tutorial planner does reason about the attribute. For example, the attributes of Plan 1 are determined by the control

structure of the tutorial planner. Plan 4, on the other hand, is generated by reasoning about all four attributes.

Plan 1: The simplest plan tutors all the bugs, uses the ordering rule of starting at the top of the program and tutoring bugs in the order they appear, and simply prints out some canned text that is associated with each of the bugs in a library of bugs. Plan 1 is not especially good for several reasons, which are addressed in the descriptions of the following plans.

Plan 2: In Plan 1, all the bugs are tutored. This is not always advisable. For example BUG 5 and BUG 6 are missing guards on the output of Average and Maximum. These bugs are not very important because they only arise in the case when the user of the program starts it up and then immediately enters the sentinel. In addition, the idea of making sure undefined values are reported as such is not a key concept in an introductory programming course, especially when compared to other problems the student who wrote the program apparently has. Therefore, the first improvement that can be made on the first tutorial plan is to decide which bugs to tutor. In Plan 2, BUG 5 and BUG 6 are not tutored.

Plan 3: In Plan 2, the bugs are still tutored in the order in which they appear in the program. In Plan 3, however, the tutor tries to eliminate aspects of the program that make it hard to tutor the student because the behavior of the program is hard to understand. One problem that makes the behavior of the program hard to understand is BUG 3, the missing statement to read a new value of RAINFALL in the loop. This causes an infinite loop that may confuse the student. In Plan 3, the tutor picks BUG 3 to tutor first

with the idea that cleaning up the output of the loop will make it easier to work on the bug that is the real problem, the student's misunderstanding about how loops work, which leads to the misplaced update below the loop.

Plan 4: Sometimes getting a student to think about one bug facilitates the tutoring of a second bug. In Plan 3, no attempt is made to use one bug to facilitate the tutoring of another bug. For example, BUG 4, the misplaced update of TOTALRAINFALL, suggests that the student does not understand the scope of a WHILE loop. Perhaps the student thinks that any sequence of BEGIN/END blocks following a WHILE statement is executed in the scope of the WHILE statement. If the tutor could focus the student's attention on the true behavior of the loop before tutoring the misplaced update, that would lead the student to see what the loop actually does and maybe even to identify BUG 4 without the tutor's help. Plan 4 thus is different from Plan 3 in two ways. First, the tutor uses BUG 2, the unconditional update of RAINDAYS, a bug that is easy to understand and to fix, as a way to focus the student's attention on the loop: these two bugs are therefore grouped together. Second, instead of just reporting to the student some canned text associated with the bug, the tutor uses a strategy of asking the student to simulate several iterations of the loop and to keep track of the value of RAINDAYS when RAINFALL is positive on some iterations and 0 on others. Finally, if there is any time left in the tutoring session, and it has gone well, the tutor might work on the unguarded outputs with the idea of teaching the student about the idea of boundary conditions.

In short, there is a range of complexity for plans that could be used to organize the tutoring for a student who makes several bugs. Some plans are very simple and do not require any reasoning about issues such as bug importance or relationships among bugs; others must be constructed by tutorial planners that can reason about various kinds of knowledge, such as the importance of bugs, strategies to use with bugs, and so forth. The following section describes planning mechanisms that generate the foregoing plans.

4.3. Tutorial Planners

This section describes the four algorithms that generate the four tutorial plans described in the previous section and indicates the types of knowledge that are required by the tutorial planners. The algorithms, and the types of knowledge required for them to operate, are shown in Figure 4.2. As will be seen, the algorithms that generate the more sophisticated plans are correspondingly complex.

Tutorial Planner 1: Tutorial Planner 1 does not appeal to any knowledge to generate its plan. All the "knowledge" of the planner resides in its control structure. The planner simply starts with the first bug in the student's program, associates the appropriate canned text with the bug, and continues until there are no more bugs in the student's program. The control structure for this planner is shown in Figure 4.2. Alongside the description of the control structure is a list of the knowledge required for the tutorial planner to operate. As can be seen, Tutorial Planner 1 does not

require any knowledge and the tutorial plan is generated as a direct result of its control structure.

Tutorial Planner 2: As the algorithm for Tutorial Planner 2 shows, this tutorial planner makes a single decision about each bug in the student's program. This decision is whether the bug is sufficiently important to tutor. In the case of the program shown in Figure 3.6, the decision about bug importance requires the tutor to reason about whether the concepts that the student must understand to avoid the bug in the future are important enough to justify tutoring. For example, the tutorial planner decides that the concept of unlikely boundary conditions, implicated by BUG 5 and BUG 6, are not important enough to justify tutoring. On the other hand, the student's problem with the concept of the scope of a loop is important enough to justify tutoring.

Tutorial Planner 3: Tutorial Planner 3 makes two decisions. In addition to deciding about the importance of bugs, Tutorial Planner 3 determines which bugs interfere with the tutoring of other bugs. In the case of the program shown in Figure 3.6, the tutorial planner decides that the infinite loop caused by the missing READ(RAINFALL); statement could make it difficult to tutor the misunderstanding about the scope of loops that leads to the misplaced update of TOTALRAINFALL. Thus, the tutor decides to clean up the infinite loop before tackling the misplaced update.

Tutorial Planner 4: In addition to bug importance and tutoring interference, Tutorial Planner 4 reasons about several additional issues. First, Tutorial Planner 4 reasons about which bugs are hard to tutor and, therefore, how to use bugs that are easy

to tutor to lead up to bugs that are hard to tutor. Second, Tutorial Planner 4 reasons about which strategies are effective in tutoring bugs. As the description of the control structure for Tutorial Planner 4 shows, this introduces considerable complexity. The tutor first reasons about how to use easy bugs to lead up to important bugs that are hard to tutor. Next the tutor must decide what strategies to use for bugs so that 1) the bugs that interfere with tutoring hard bugs can be quickly fixed and 2) the strategies for bugs that are intended to lead up to the hard, important bugs are compatible with the strategies for the hard, important bugs. Then the tutor reorders bug groups so that the interfering bugs come early and reorders bugs within bug groups so lead in bugs come before hard bugs.

TUTORIAL PLANNER CONTROL STRUCTURES	KNOWLEDGE USED
Tutorial Planner 1	
For each bug in BUGS	
Get canned text for first bug	NONE
Add bug and canned text to TUTORIAL_PLAN	
Tutorial Planner 2	
For each bug in BUGS	
If first bug is important Then	IMPORTANCE
Begin	
Get canned text for first bug	
Add bug and canned text to TUTORIAL_PLAN	
End	
Tutorial Planner 3	
For each bug in BUGS	
Begin	
If first bug is important Then	
Begin	IMPORTANCE
Get canned text for first bug	TUTORING INTERFERENC
Add bug and canned text to TUTORIAL_PLAN	
End	
End	
Reorder bugs in TUTORIAL_PLAN such that:	
If fixing BUG-X makes BUG-Y easier understand	
Then put BUG-X before BUG-Y in TUTORIAL_PLAN	
Tutorial Planner 4	
Begin	
Make groups of bugs in BUGS such that:	IMPORTANCE
easy bugs lead up to	TUTORING INTERFERENC
important bugs that are hard to tutor	TUTORING DIFFICULTY
Add groups of bugs to TUTORIAL_PLAN	LEADING TO HARD BUGS
Select strategies for groups in TUTORIAL_PLAN such that:	STRATEGIES
clean up bugs can be quickly fixed	
strategies for lead in bugs compatible with	
strategies for important bugs	
Reorder between bug groups in TUTORIAL_PLAN such that:	
clean up bugs are before important bugs	
Reorder within bug groups in TUTORIAL_PLAN such that:	
lead in bugs come before important bugs	
Mark groups of unimportant bugs "tutor if time remains"	
End	

Figure 4.2: Control Structures and Knowledge for Planners

4.4. Tutorial Plans, Tutorial Planners, and Variability

Chapter 1 showed that two types of variability -- inter-subject variability and contextual variability -- can affect how a tutor approaches a student's bugs i.e., the plan that the tutor formulates. The purpose of this section is to identify key relationships between the two types of variability and the characterization of tutorial plans and tutorial planners presented in this chapter. This discussion compares the impact of the two types of variability on Tutorial Planner 1, the simplest tutorial planner described in this chapter, and and Tutorial Planner 4, the most complex tutorial planner treated in this chapter.

It is somewhat ironic that simple tutorial planners, and the simple plans that they generate, are less affected by the two kinds of variability than more complex tutorial planners and plans. Tutorial Planner 1, for example, is adversely affected by variability only to the extent that its library of bugs is incomplete. Because Tutorial Planner 1 uses the same ordering strategy -- textual order -- for all bugs, and the same tutorial strategy -- printing out canned text -- the problems of 1) grouping together non-consecutive bugs, and 2) using different strategies for the same bug depending on the other bugs it appears with, do not arise. Thus, if the bugs for which Tutorial Planner 1 must develop a tutorial plan are represented in its bug library, Tutorial Planner 1 is completely unaffected by either type of variability.

For example, Tutorial Planner 1 would treat BUG 1 in Figure 1.5 in the same way regardless of the other bugs the student made. In both cases, Tutorial Planner 1 would address BUG 1 first and output the same canned text. Tutorial Planner 1 cannot reason that BUG 1 in the second program in Figure 1.5 should be addressed with the other bugs in the plan to find the maximum of RAINFALL and not with the other missing initializations. Indeed, Tutorial Planner 1 has no understanding of programming plans, similar bugs, alternative strategies, and so forth. Thus, the rigid control structure of Tutorial Planner 1 reduces the problem of variability to the problem of how many bugs it has in its bug library.

At the other extreme, Tutorial Planner 4 is affected by both inter-subject variability and context variability. Inter-subject variability affects Tutorial Planner 4 because Tutorial Planner 4 does not rely on a rigid control structure and a library of bugs with canned advice for the student. Rather, Tutorial Planner 4's control structure uses many types of knowledge to create a tutorial plan: Tutorial Planner 4 uses a representation of the task specification in terms of its goals and associated plans; a representation of various aspects of the curriculum, such as the importance of e.g., iteration; knowledge about what makes bugs hard to tutor; knowledge about appropriate groupings of bugs, appropriate orderings of bugs, and so forth. Thus, Tutorial Planner 4's ability to handle inter-student variability depends on far more than a simple, fixed library of bugs with canned text designed to help the student with the bugs.

Context variability requires Tutorial Planner 4 to reason about the relationship of a particular bug in a student's program to the

other bugs in the student's program. The reasoning that Tutorial Planner 4 may have to do to handle context variability can depend on the task specification network; the types of effects of different bugs; plan components that bugs have in common, and so on. For example, the two programs shown in Figure 1.5 contain in common, among others, BUG 1, the missing initialization of MAXRAIN. As the discussion in Chapter 1 suggested, BUG 1 fits differently into the tutorial plan depending upon whether it is 1) similar in type to other bugs (the missing initializations shown in the first program in Figure 1.5) or part of the same task goal (the incomplete, incorrect plan to find MAXRAIN shown in the second program in Figure 1.5).

4.5. Chapter Summary

In sum, there are many possible tutorial plans for the same set of multiple bugs. The possible plans range from very simple to quite sophisticated. The control structure required to generate sophisticated tutorial plans is more complex than the control structure required to generate simple tutorial plans. In addition, the generation of more complex tutorial plans requires more knowledge than the generation of simple tutorial plans. The increased knowledge required to generate sophisticated tutorial plans requires a tutorial planner to handle both inter-student variability and context variability. A primary goal of the dissertation is to construct a tutorial planner that produces tutorial plans that are like those produced by experienced human tutors. The next chapter

describes TP, the tutorial planner implemented in a computer program, that generates human-like tutorial plans for novices' multi-bug computer programs.

Chapter 5

THE TUTORIAL PLANNER: TP

5.1. Introduction

TP is the computer program that generates tutorial plans for multiple bugs like the tutorial plans that experienced human tutors generate. TP's generation of tutorial plans is guided by its rules for making the five Tutorial Planning Decisions described in Chapter 1 and discussed in detail in Chapter 3. The processing mechanism that makes the five Tutorial Planning Decisions uses tutorial knowledge in the four Knowledge Categories described in Chapter 2 and Chapter 3.

The main goal of this chapter is to describe TP. The focus of this chapter is on 1) the way TP represents the tutorial planning knowledge described in Chapter 2 and Chapter 3 and 2) the control structure that governs how TP generates tutorial plans for multiple bugs. This chapter contains five main sections.

- First, there is a general overview of TP. The general overview is intended to provide a synopsis of TP without burdening the reader with too many details.
- Second, TP's knowledge is described in detail. This section focuses on TP's knowledge about 1) the domain of novice programming, 2) bugs, and 3) tutorial planning rules.
- Third, TP's representation of tutorial plans is described.

• Fourth, the architecture and processing flow of TP are described.

• Fifth, the main lessons learned from constructing the TP are identified.

The reader who is not interested in the details of TP only needs to read the general overview section and, possibly, the final section, which contains examples of tutorial plans for multiple bugs.

5.2. General Overview of TP

This section provides an overview of TP. The goal is to give the reader a general picture of 1) TP's rule knowledge about tutorial planning, 2) TP's representation of tutorial plans, and 3) TP's architecture. The presentation of topics in this section mirrors the structure of the remainder of the chapter, which works out the details of the topics presented in this overview section.

TP constructs a tutorial plan by making the five Tutorial Planning Decisions. TP represents each of the five Tutorial Planning Decisions as a Tutorial Planning *Goal.* TP uses a simple rule-based planning architecture in which each of the five Tutorial Planning Goals is achieved by applying rules specific to that Tutorial Planning Goal. As the rules associated with each Tutorial Planning Goal fire, the Tutorial Plan is gradually built up. The Tutorial Plan is specified when all the Tutorial Planning Goals have been achieved. For example, the first Tutorial Planning Decision, how to group bugs together, has a corresponding Tutorial Planning Goal in TP. When

that Tutorial Planning Goal is achieved, the grouping decisions are made.

Figure 5.1 depicts the architecture of TP. The level of detail in the figure corresponds to the description of TP given in this general overview. The figure shows the three primary components of TP. Each component is indicated by the underlined words along the left side of the figure.

• INPUT: The input to TP consists of two components: 1) a list of descriptions of the student's bugs and 2) TP's Knowledge Base, which comprises the Knowledge Categories of Student, Domain, and Bugs, and Teaching. The first three of these Knowledge Categories are described in Chapter 2. Knowledge Category 4 is described in Chapter 3. Each bug in the student's program is described by a record structure with the fields identified below.

• PROCESSING: TP's processing is controlled by a Rule Applier which executes rules stored in the Knowledge Base. A separate set of rules is responsible for achieving each of the Five Tutorial Planning Goals. The Tutorial Planning Rules depend upon the knowledge types in Knowledge Categories 1, 2, and 3.

• OUTPUT: TP's output is the Tutorial Plan for multiple bugs. The Tutorial Plan specifies groups of bugs, how much attention to give to each bug, the order in which to tutor bugs, the objectives to be achieved in the tutoring session, and the strategies to be used to achieve the objectives.

In short, TP uses a straightforward rule-based architecture to build a Tutorial Plan for multiple bugs. The following describes each of the three components of Figure 5.1. The discussion of the

output -- the Tutorial Plan -- precedes the discussion of the processing mechanism because the discussion of the processing mechanism describes actions performed on the Tutorial Plan i.e., the output.

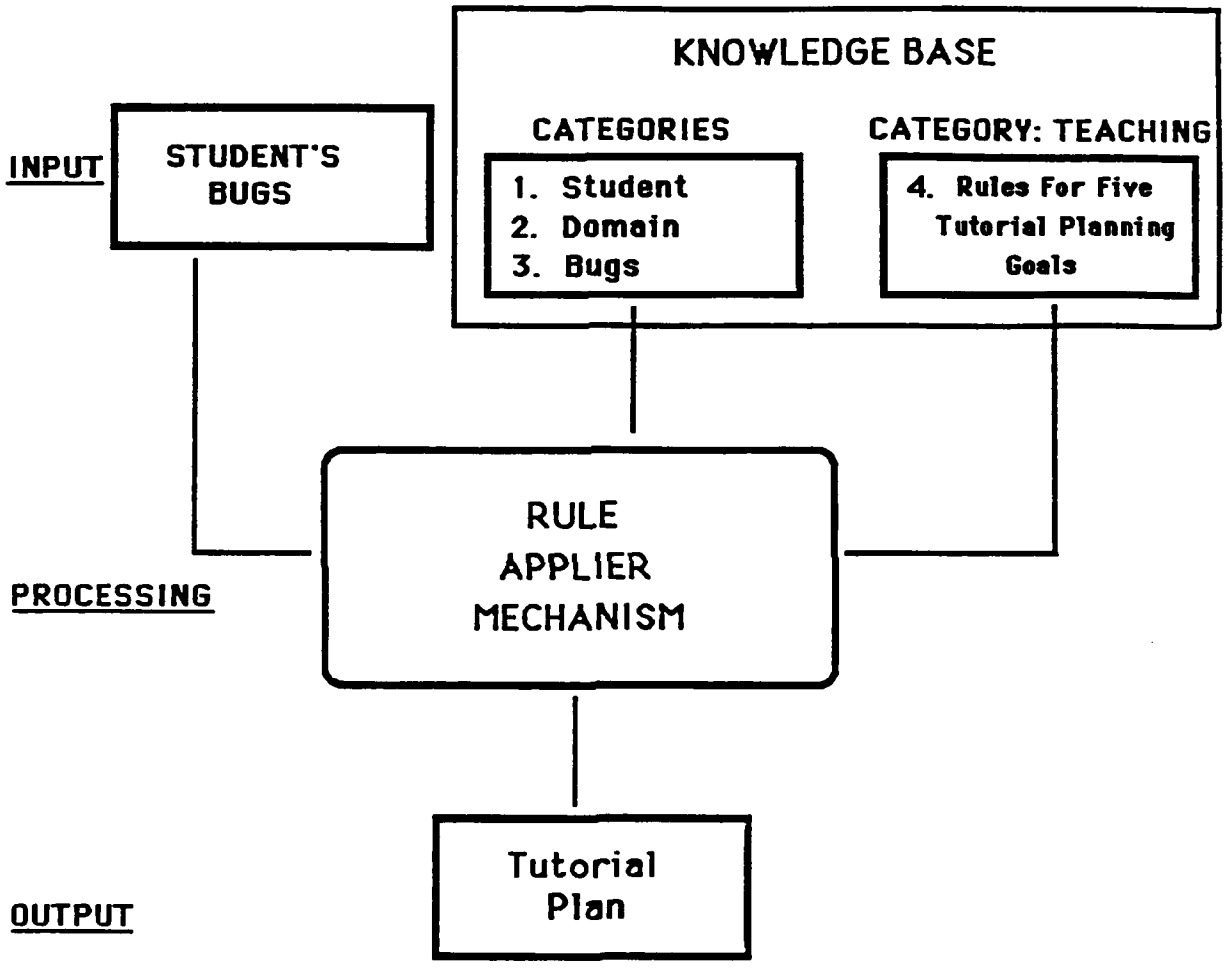


Figure 5.1: TP Input/Processing/Output

5.2.1. Tutorial Planning Knowledge: INPUT

TP's tutorial planning knowledge resides in TP's Knowledge Base, as shown in Figure 5.1. TP's Knowledge Base contains the representation of the knowledge types in Knowledge Categories 1, 2, 3, and 4. The knowledge types in the first three Knowledge Categories are represented declaratively. Knowledge Category 4, TP's knowledge about teaching, is represented as rules about how to construct tutorial plans.

5.2.1.1. Knowledge Categories 1, 2, and 3

TP's input has three categories of knowledge. The three Knowledge Categories, discussed in Chapter 2, are knowledge about students, knowledge about the domain of programming, and knowledge about bugs. The purpose of this section is to describe those three Knowledge Categories.

• Knowledge Category 1: TP's knowledge about students is represented in record structures that have exactly the features identified in the section in Chapter 2 which described Knowledge Category 1, knowledge about students.

A record for an individual student contains fields for ability, motivation, concepts mastered, and skills mastered. TP can be initialized with different students simply by changing the values in the fields of the student record. For example, it is possible to add to, and delete from, a student record programming skills such as

debugging simply by changing the values in the field corresponding to programming skills mastered. Figure 5.2 shows a student record.

The Tutorial Planning Rules, described below, refer to information in the student record when constructing the tutorial plan. For example, the student shown in the Figure 5.2 has high ability and high motivation. Therefore, the rules that achieve Tutorial Planning Goal 3, and make the ordering decision, would decide to start the tutoring session with a difficult bug.

• Knowledge Category 2: TP's knowledge about the programming domain represents the four knowledge types shown in Figure 2.3 under Category 2:

• Task Specifications are represented as a partially ordered graph of goals which represent precondition relationships. Each node in the graph corresponds to a task specification goal, such as initializing the counter for RAINDAYS. Each node in the task specification graph identifies a goal, the preconditions for the goal, and the actions that will achieve the goal. An example of a task specification goal is shown in Figure 2.5.

• Programming Plans are represented as record structures which have the attributes discussed in Chapter 2. Figure 2.6 shows several important plans that novice programmers learn. TP has two kinds of knowledge about programming plans. First, the importance of a programming plan is either high, moderate, or low. Second, there may be alternative ways to achieve the same plan in the code: Figure 2.7 shows three ways to write a running total loop plan. TP uses the representation of programming plans for two main purposes. First, the importance of the programming plan is used in

cc tion with any bugs the student has made in the plan to decide wh it should be addressed in the tutoring session. For example, a trivial bug in a low importance plan is less likely to be tutored than an important bug in a high importance plan. Second, the alternative ways of writing the same plan can be used if the tutor formulates the objective to teach the student a better way to achieve one of the student's goals.

• Programming Concepts are represented in a graph that is partially ordered by a knowledge precondition relation. In addition, each concept has an importance value. There are approximately 10 nodes in the graph, reflecting the concepts shown in Figure 2.8. A piece of the concept graph is shown in Figure 5.3. This example shows that the important concept of iteration has two important knowledge precondition concepts: sequencing and scoping. Rules that decide on the ordering of bugs in the Tutorial Plan refer to the graph of Programming Concepts to decide which bugs point to co epts that should be tutored before the concepts pointed to by other bugs.

• Programming Skills are represented in terms of their subskills. For example, the skill of Reorganizing, illustrated in Figure 2.9, requires the programmer to 1) identify a segment of the program that is not cleanly written, 2) reanalyze the plans selected to achieve the goals of the segment, 3) select alternative plans, and 4) instantiate the alternative plans in the code. For example, a student whose Rainfall Program unnecessarily re eats the test for valid data (RAINFALL >= 0) in the main loop mi : decide to collapse all the actions guarded by the test into a single block covered by one

test for RAINFALL >= 0. TP does not represent precondition relations among programming skills.

• Knowledge Category 3: TP's knowledge about bugs, Knowledge Category 3, is encoded in terms of the values that can fill the slots of a record structure that represents a bug. A bug record structure has a slot for 1) the cause of the bug, 2) the effects of the bug, 3) the type of the bug, and 4) the location of the plan in which the bug occurs. Each field of a bug record can have any of the values described in Chapter 2. For example, Figure 5.4 shows a description of the misplaced update of TOTALRAIN, BUG 5 in Figure 1.1. The description of the bug has a field for its name, which has no meaning beyond serving as a unique identifier. The cause of the bug is a misconception about the scoping of a WHILE loop; the effects of the bug are to give wrong values to TOTALRAIN and AVERAGERAIN; the type of the bug is a misplaced running total update; and the location of the bug is below the main loop.

The five fields show in Figure 5.4 are the main attributes of bugs in TP. Bug records actually have more fields than these five fields; they are described later in the chapter.

In summary, TP uses declarative representations for the knowledge types in the first three Knowledge Categories: Students, the Domain, and Bugs. Legal values for the fields in the declarative representations are described in Chapter 2. A fuller discussion of the implementations of the knowledge types in the first three Knowledge Categories is given later in the chapter.

NAME:	Chris
ABILITY:	High
MOTIVATION:	High
CONCEPTS MASTERED:	Loop, If-Then-Else, ...
SKILLS MASTERED:	Boundary Testing, Test Data Generation, ...

Figure 5.2: A Student Record

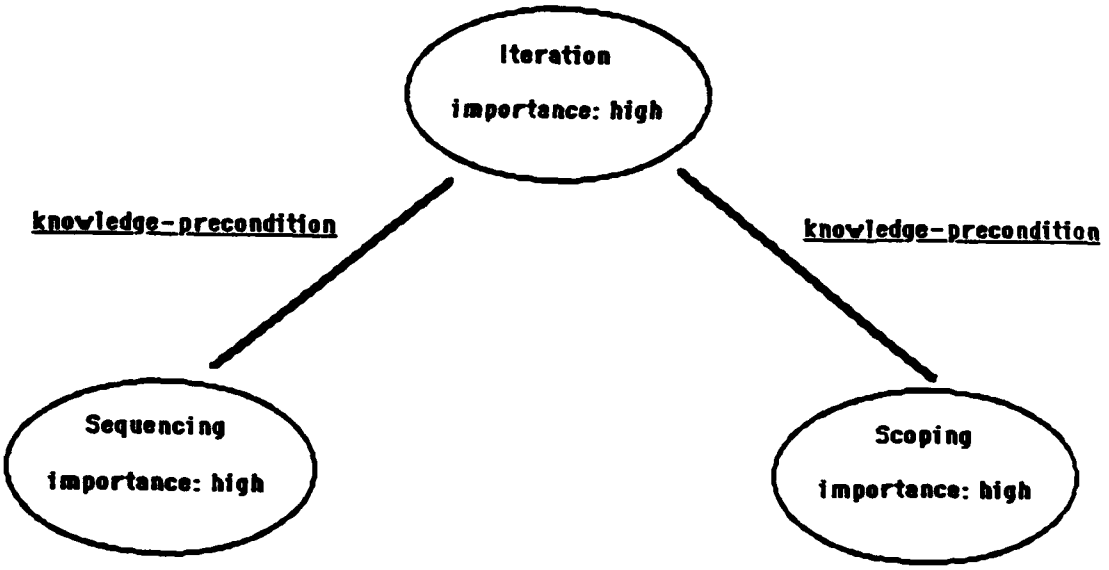


Figure 5.3: Portion of Curriculum Graph

NAME:	UPDATE-TOTALRAIN-BELOW-LOOP
CAUSE:	MISCONCEPTION : SCOPING WHILE LOOP
EFFECTS:	WRONG VALUE : TOTALRAIN AVERAGERAIN
TYPE:	MISPLACED RUNNING TOTAL UPDATE
LOCATION:	BELOW MAIN LOOP

Figure 5.4 : An Example Bug

5.2.1.2. Knowledge Category 4: Teaching Knowledge

The rightmost box in the picture of TP's Knowledge Base (Figure 5.1) depicts knowledge about teaching. The bulk of TP's knowledge about teaching consists of Tutorial Planning Rules that achieve the five Tutorial Planning Goals. A separate set of Tutorial Planning Rules is associated with each Tutorial Planning Goal. TP constructs tutorial plans for multiple bugs by running the Tutorial Planning Rules associated with each of the five Tutorial Planning Goals.

Tutorial Planning Rules, whose precise form is described in a later section of the chapter, are straightforward IF-THEN rules which are run by a rule interpreter, also described in a later section of this chapter. The Tutorial Planning Rules are partitioned into sets that correspond to the five Tutorial Planning Goals.

TP's Knowledge Base contains approximately 100 Tutorial Planning Rules. Each row of Figure 5.5 shows 1) a Tutorial Planning

Goal, 2) an example Tutorial Planning Rule that corresponds to the Tutorial Planning Goal and 3) the number of Tutorial Planning Rules that are associated with the Tutorial Planning Goal. For example, there are approximately 25 rules associated with the first Tutorial Planning Goal, the grouping goal. One such rule, shown in Figure 5.5, constructs bug groups in which one of the bugs is made in the attempt to repair a second bug.

In this overview section, abbreviated forms of Tutorial Planning Rules are shown -- complete rules are shown in a later section of this chapter. TP also has knowledge about how to recover from tutorial planning failures during tutorial plan execution. This topic is addressed in a later chapter.

Tutorial Planning Goal	Example Rule (abbreviated form)	Number of Rules
1. Group Bugs	IF BUG-1 is Repair for BUG-2 THEN Group together	25
2. YES-NO	IF Bug Cause is Serious THEN Definitely Tutor	20
3. Ordering	IF BUG-1 makes BUG-2 Easy to Tutor THEN BUG-1 before BUG-2	15
4. Objectives	IF Cause Bad Misconception THEN Fix Misconception	25
5. Strategy	IF Objective is Make Student Aware of Effects of Bug THEN Use Hand Simulation	20

Figure 5.5: Tutorial Planning Goals and Rules

5.2.2. Tutorial Plans: OUTPUT

TP's tutorial plans represent the results of achieving the five Tutorial Planning Goals. A Tutorial Plan consists of a list of bug groups. The bug groups:

- are ordered
- are labeled according to whether they should be addressed or not
- have objectives identified that should be achieved in addressing the bug group
- have strategies identified to achieve the objectives.

Figure 5.6 shows the main attributes of a bug group. Figure 5.7 shows two bug groups constructed from a subset of the bugs in Figure 1.1, namely BUG 3, BUG 4, BUG 5, BUG 8, and BUG 9. For the purpose of illustrating the attributes of bug groups and the attributes of tutorial plans, the figure shows a simplification of the actual bug groups generated by TP when it is confronted with all the bugs in Figure 1.1.

BUG GROUP 1 contains the two missing output boundary guards. This bug group should be addressed only if there is time to do so and, if addressed, should be used to make the point that unexpected boundary conditions can arise in even the simplest programs. BUG GROUP 2 contains the hard bug in the program, namely BUG 5, the misplaced update of TOTALRAIN that was probably caused by a misconception about the scoping of loops. The two other bugs in BUG GROUP 2 play the roles of 1) cleaning up the program to make the

hard bug easier to approach (BUG 4) and 2) focusing the student's attention on the behavior of the main loop.

Figure 5.7 shows a tutorial plan for BUG GROUP 1 and BUG GROUP 2. The tutorial plan specifies the order in which the two bug groups should be addressed and the emphasis that should be placed on each of them. As the figure shows, BUG GROUP 2 should be addressed first followed by BUG GROUP 1. In addition, the tutorial plan makes explicit the fact that BUG GROUP 2 should receive the most attention.

---

1. **BUGS IN GROUP**
2. **GROUP TYPE**
3. **YES-NO**
4. **OBJECTIVES**
5. **STRATEGIES**

Figure 5.6: Attributes of Bug Groups

1.	**BUGS IN GROUP:**	BUG 8, BUG 9 Missing Output Guards of AVERAGERAIN and MAXRAIN
2.	**GROUP TYPE:**	Same Plan Component Housekeeping
3.	**YES-NO:**	Tutor if time
4.	**OBJECTIVES:**	Make student aware of bugs Emphasize boundary conditions
5.	**STRATEGIES:**	Ask for simulation of test case of no input Compare actual with correct output

BUG GROUP 1

1.	**BUGS IN GROUP:**	BUG 3, BUG 4, BUG 5 Unconditional Update RAINDAYS No READ(RAINFALL); in loop Misplaced update TOTALRAIN
2.	**GROUP TYPE:**	Hard bug (with clean up and lead in)
3.	**YES-NO:**	Definitely Tutor
4.	**OBJECTIVES:**	First: Clean up infinite loop BUG 4 Second: Lead in to problem with loop Third: Fix misconception about scoping
5.	**STRATEGIES:**	Point out missing READ(RAINFALL); Simulation to detect unconditional update RAINDAYS

BUG GROUP 2

Figure 5.7: Two Bug Groups

ORDER: 1. BUG GROUP 2
2. BUG GROUP 1

EMPHASIS: BUG GROUP 2
More important than
BUG GROUP 1

Figure 5.8: A Simple Tutorial Plan

```
Initialize TUTORIAL-PLANNING-GOAL-QUEUE with Tutorial Planning Goals
For GOAL in TUTORIAL-PLANNING-GOAL-QUEUE Do
   Begin
      Initialize Rule Applier with TUTORIAL-PLANNING-RULES for GOAL
      For T-P-RULE in TUTORIAL-PLANNING-RULES
         Attempt  T-P-RULE
   End
```

Figure 5.9: The Top Level of the Tutorial Planning Algorithm

5.2.3. Architecture & Processing Flow: PROCESSING

The purpose of this section is to provide a high level view of TP's architecture and processing mechanism. Details of the processing mechanism, such as its goal queue and rule selection mechanism, are discussed in a later section.

There are no subtleties in TP's architecture. As Figure 5.1 shows, TP's design is a straightforward combination of two components: 1) a Knowledge Base that contains the knowledge described in Chapter 2 and Chapter 3 and 2) a Rule Applier.

The central feature of TP's processing mechanism is the Rule Applier. The Rule Applier uses a simple algorithm that executes the Tutorial Planning Rules associated with each Tutorial Planning Goal. The high level description of TP's algorithm is shown in Figure 5.9:

- First, the processing mechanism initializes a queue of Tutorial Planning Goals with the five Tutorial Planning Goals.
- Second, the Tutorial Planning Rules associated with the current Tutorial Planning Goal are passed to the Rule Applier. For example, if the current Tutorial Planning Goal were to select strategies for the bug groups in the Tutorial Plan, then the approximately 20 Tutorial Planning Rules that know how to select tutorial strategies would be passed to the Rule Applier.
- Third, the Rule Applier executes the Tutorial Planning Rules for the current Tutorial Planning Goal. Each Tutorial Planning Rule has an IF part and a THEN part. The IF part of each rule is a conjunct of disjuncts or conjuncts. The IF parts of the rules typically perform tests on 1) the student's bugs, 2) the student's ability, and 3) current attributes of the Tutorial Plan.

For example, a rule might ask 1) if the student has made several bugs in the same minor plan component, 2) if the student has high ability and 3) the tutorial plan currently contains no serious bugs. These tests, and the way in which they are represented, are described in detail in a later section.

The THEN part of each rule is a conjunct of actions to be taken when the criteria stated in the IF part are met. Each Tutorial Planning Rule that fires makes an addition to, or modification of, the Tutorial Plan.

For example, if the Tutorial Plan specifies the objective of making the student aware of the effects of a particular bug, such as a missing guard to protect the calculation of AVERAGERAIN from division by zero, and the current Tutorial Planning Goal is to select strategies, then the strategy selection rules might add a link between the bug and the strategy of hand simulation of the program. During tutoring the tutor might ask the student to simulate the calculation of AVERAGERAIN when the user does not enter any valid RAINFALL data.

- This process continues until there are no more Tutorial Planning Goals.

Additional data structures and components of the processing mechanism support TP's tutorial planning activity. Because they are not key to understanding TP's general character, they are described in a later section of this chapter.

5.2.4. Summary

TP generates tutorial plans for multiple bug programs written by novice Pascal programmers. The input to TP is a list of the student's bugs and a knowledge base that contains tutorial planning knowledge. The output of TP is a Tutorial Plan that results from making the five Tutorial Planning Decisions. Each of the Tutorial

Planning Decisions is implemented in TP as a Tutorial Planning Goal with associated Tutorial Planning Rules that are responsible for achieving that Tutorial Planning Goal. TP's processing is performed by a straightforward rule application mechanism that executes the Tutorial Planning Rules.

5.3. TP's Knowledge

The purpose of this section is to describe further the data structures that encode the knowledge that TP uses to generate tutorial plans. Several of the simpler data structures e.g., the student, were described earlier. Others, such as the representation of task specification goals, were described very generally. The focus of this section is therefore on TP's knowledge about the domain i.e., task specifications, bugs, and tutorial planning knowledge.

5.3.1. Knowledge about the Domain

The previous sections have described programming concepts and skills, which have a simple representation. Two types of knowledge about the domain that have more complex representations are task specifications and programming plans. This section describes those two types of knowledge

5.3.1.1. Task Specifications

Figure 5.10 shows an example of the record structure that represents each goal in a task specification. Each task goal is a record in a network connected by two slots in the task goal representation, namely the supergoals slot and the subgoals slot. In the example shown in Figure 5.10, the attributes of the goal to calculate TOTALRAIN are shown.

Each task goal has eight slots. The commented text to the side of the slots gives the fillers for the goal to calculate TOTALRAIN. For example, the objects involved in the calculation are RAINFALL and TOTALRAIN; the subgoals are, for example, to input RAINFALL; the action to be taken is to sum RAINFALL into TOTALRAIN; and a plan for the goal is the running total loop plan. The entire set of goals for the Rainfall Task and for the Reformatting Task are shown in Figure 6.1, Figure 6.2 and Figure 6.3.

5.3.1.2. Programming Plans

Figure 5.11 shows the representation for programming plans. Each programming plan has five attributes: a name; an action; objects on which the action is performed; constraints; and location. The example shown in Figure PROGRPLANREP is the initialization of DAILYRAINFALL. The action to be taken in the initialization is an assignment. The object to be assigned is DAILYRAINFALL. The single c -aint on the assignment is that it not equal the sentinel. The location of the initialization is before the main loop. Each of the

approximately 60 programming plans in TP's knowledge base has the format shown in Figure 5.11.

5.3.2. Knowledge about Bugs

Each bug in TP is represented as a record structure with 20 attributes. The example in Figure 5.12 shows the bug in which the student has spuriously initialized DAILYRAIN after reading the initial value of DAILYRAIN. The spurious initialization destroys the first value of DAILYRAIN that was read in just before the spurious initialization.

The representation of each bug contains, among others, attributes about the task specification component the bug is in; parent plans of the plan in which the bug is located; the objects associated with the bug; and the operation which is performed on the objects. In addition, each bug record indicates the focality, type, and location of the bug, the cause of the bug, what the student's intent was that led to the bug, the effects of the bug and whether the student is aware of them, the actual and correct code, the tutorial difficulty of the bug and its importance, and five kinds of information that are used for housekeeping during the execution of the tutorial planning program.

Each of the bugs shown in Figures 6.4 and 6.5, and Figure 6.6 is represented in the form shown in Figure 5.12.

```
(defstruct TASKGOAL
      goal              ; e.g., (calculate TOTALRA  l)
      goal-type               ; e.g., calculate
      objects           ; e.g., RAINFALL, TOTALRAIN
      supergoals        ; e.g., (calculate AVERAGERAIN)
      subgoals          ; e.g., (input RAINFALL)
      action            ; e.g., (sum-into RAINFALL TOTALRAIN)
      plans-for         ; e.g., running-total-loop
      importance        ; e.g., high
      )
```

Figure 5.10: Representation of a Task Specification Goal

```
(defstruct PROGPLAN
      name              ; initialize-DAILYRAINFALL
      action            ; e.g., assign
      objects           ; e.g., DAILYRAINFALL
      constraints       ; e.g., (NotEqual Sentinel
      location          ; e.g., (before main-loop
      )
```

Figure 5.11: Representation of a Programming Plan

```
(make-BUG
        :NAME           'init-DailyRain-0-clobbers-readrainfall
        :SPEC           '((implicit (init DailyRain)))
        :PARPLANS       '(calc-TotalRain)
        :PLANCOMP       'init
        :OBJECTS        '(DailyRain)
        :OPERATION      '((Assign DailyRain 0))
        :FOCALITY       'low
        :TYPE           'spurious
        :LOC            '((after (init DailyRain)))
        :CAUSE          '((plan-pollution))
        :STUD-INTENT    '((init DailyRainfall))
        :EFFECTS        '((Stud-Unaware (wrong-value TotalRain))
                        (Stud-Unaware (clobber-value-before-use TotalRain)))

        :CODE '((BEGIN)
            (KEYSECTION READLN (DailyRain)#\;
                    DailyRain #\: = 0 #\;)
            (WHILE (DailyRain <> 99999))
            (BEGIN)
            (TotalRain := TotalRain + DailyRain #\;)
            (END #\;))

        :CODE-SHOULDBE '((BEGIN)
                (KEYSECTION READLN (DailyRain)#\;)
                (WHILE (DailyRain <> 99999))
                (BEGIN)
                (TotalRain #\: = TotalRain + DailyRain #\;)
                (END #\;))

        :TUTDIFF        'low
        :IMPORTANCE     'medium
        :BUGGROUP       nil
        :USEDAS         nil
        :TUTORED?       nil
        :RULESFIRED     nil
        )
)
```

Figure 5.12: Representation of Bugs

5.3.3. Tutorial Planning Rules

The tutorial planning rules that implement each of the tutorial planning goals in TP are simple production-like objects. The rules are stored in packages each of which corresponds to a Tutorial Planning Goal. Each rule has a set of conditions which must be met in order for the rule to fire and a set of actions to be taken if the rule fires. Figure 5.13 shows an example rule. The rule shown in Figure 5.13 creates a bug group of type GOOD REPAIR if it encounters two unassigned bugs where one of the pair is a good repair for the other. Note that the action of the rule 1) creates a new group 2) adds the two bugs to a newly made group and 3) adds the attributes for group type to the descriptors for the group. The three attributes added to the group -- Bug Dependency Program Behavior and Good Repair-- correspond to the position of this group type in the tree of bug group types shown in Figure 3.1.

The rules in TP are written with one level of syntactic sugar. The list of conditions for each rule, headed by IF, is expressed as any combination of arbitrarily nested conjuncts and disjuncts. The list of actions for each rule, headed by THEN, can be arbitrary lisp code. A lisp function takes each rule expression and creates a lambda expression that is run by the rule interpreter which is part of TP's control mechanism.

```
(RULEMAKE=Make2BugRule 'GROUP-GoodRepair1.1
            '(IF
              (BUGPRED=UnequalBothUnassigned? bug1 bug2)
              (BUGPRED=Bug2GoodRepairForBug1? bug1 bug2)
              )
            '(THEN
              (BUGGROUP=MakeNewGroup (list bug1 bug2)
                        '(BugDependency ProgramBehavior GoodRepair))
              ))
```

Figure 5.13: An Example Grouping Rule

5.4. Tutorial Plans in TP

A tutorial plan in TP is an ordered list of bug groups. As Figure 5.14 shows, each bug group has nine attributes. The last two slots contain housekeeping information about whether the bug group has been addressed yet (tutored?) and which rules fired during the construction of the bug group.

The example shown in Figure 5.14 is of the good repair type in which the student has repaired the problem of adding in the sentinel to TOTALRAIN in the loop by subtracting from TOTALRAIN before performing the calculation of AVERAGERAIN. The goals associated witn the bug group are specific to the bugs in the bug group. There can be more than one goal specified for bugs in a bug group. If there is more than one goal specified, then the goal slot consists of a list of lists in which the first element in each sublist is the goal to be achieved and the remaining elements in the list are the bugs for which the goal is intended.

```
(defstruct BUGGROUP
      type                ; e.g., Good Repair
      size                ; e.g., 2
      descriptors         ; e.g., ProgBehavior BugDependency GoodRepair
      bugs                ; e.g., (add-sentinel-total back-out-sentinel)
      goals                    ; e.g., Teach-Better-Plan
      tutoradvice         ; e.g., definitely
      tutored?            ; e.g., no
      rulesfired          ; e.g., RULE-STRAT-SELECT5 ...
 )
```

Figure 5.14: Representation of a Bug Group

5.5. Architecture & Processing Flow of TP

TP uses a simple rule application architecture, as shown in Figure 5.1. The purpose of this section is to *describe, in more* detail than was provided in the general description, the processing mechanism of TP. The processing mechanism was represented in Figure 5.1 by the box labelled Rule Applier Mechanism.

5.6. TP's Architecture

Figure 5.15 shows the details of TP's architecture. The arrows connecting components in the figure represent the processing flow. The two main components of TP are 1) its planning knowledge, shown in the box labelled Tutorial Planning Knowledge, and 2) its processing mechanism, shown in the box labelled Processing Mechanism. Tutorial planning goals and tutorial planning rules have

been discussed above so the focus in this section is on the processing mechanism.

The main components of TP's processing mechanism consist of its dynamic data structures, shown in the box so labelled, and its rule mechanism, shown in the box labelled by that term. TP's control mechanism uses three dynamic data structures.

- Tutorial Planning Goal Queue: This is the list of tutorial planning goals with which TP is initialized. The initial list consists of the five Tutorial Planning Goals that correspond to the five Tutorial Planning Decisions.
- Current Planning Goal: This is the tutorial planning goal that is currently active. As each tutorial planning goal is achieved, it is removed from the tutorial planning goal queue and the next goal in the queue is made the current planning goal.
- Tutorial Plan: The attributes of the tutorial plan, as described above, are specified as the processing mechanism executes the rules corresponding to the Tutorial Planning Goals.

The rule mechanism shown in Figure 5.15 has two main components: a rule selector and a rule applier.

- The rule selector functions by identifying the current planning goal when it becomes newly active, selecting the tutorial planning rules associated with that goal, and initializing the rule applier.
- The rule applier executes whatever rules are currently on its rule list. The rule applier evaluates the lambda expressions corresponding to the rules that are associated with each Tutorial Planning Goal.

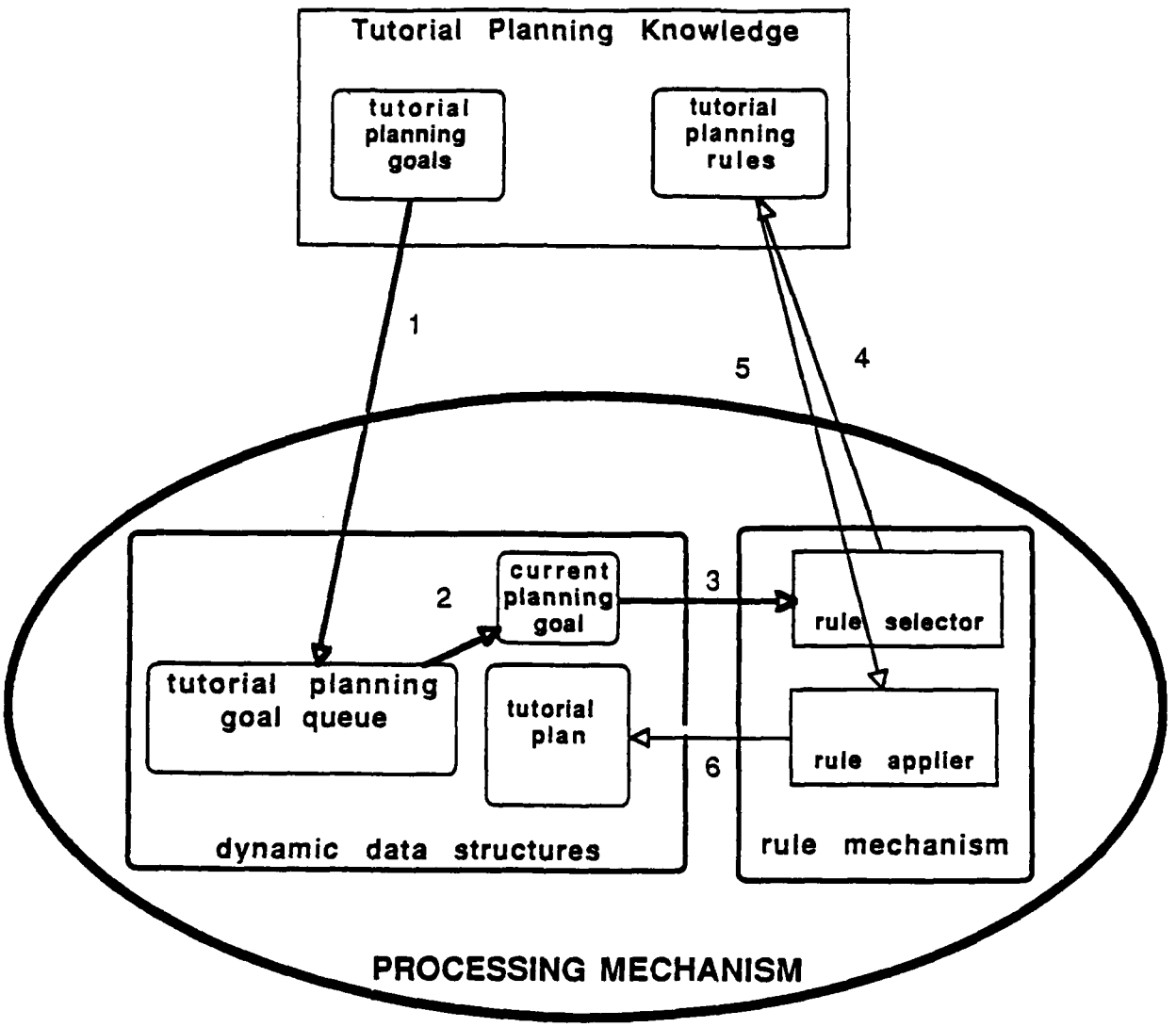


Figure 5.15: Details of TP's Processing Mechanism

5.7. TP's Processing Flow

The control algorithm for TP loops over the tutorial planning goals in the tutorial planning goal queue. As each tutorial planning goal is encountered and made the current planning goal, corresponding tutorial planning rules are executed. The processing flow generated by TP's control algorithm is represented by numbered arrows in Figure TPPROCMECH. The control algorithm for TP's processing, shown in pseudocode in Figure 5.9, has six main steps.

- Arrow 1: Initialize the tutorial planning goal queue with the tutorial planning goals.
- Arrow 2: Make the first goal on the tutorial planning goal queue the current planning goal.
- Arrow 3: Call the rule selector to choose the tutorial planning rules for the current tutorial planning goal.
- Arrow 4: Choose the tutorial planning rules for the current tutorial planning goal.
- Arrow 5: Initialize the rule applier with the tutorial planning rules for the current tutorial planning goal.
- Arrow 6: Apply the tutorial planning rules, which make changes to the tutorial plan.

In short, TP's processing algorithm is based on a simple rule interpreter that is at the center of processing. Tutorial Planning goals that represent the 5 Tutorial Planning Decisions are activated, the rules for those goals are executed by the rule applier, and

changes specified by the rules are made to the tutorial plan. TP has no provision for either backtracking or heuristic evaluation of alternative tutorial plans. The possibility of incorporating a backtracking facility in TP is discussed in Chapter 8, which explores the generality of TP as an approach to tutorial planning.

5.8. Lessons Learned

Constructing TP, the implementation of the model of tutorial planning for multi-bug novice computer programs, was a useful exercise that produced many lessons. Of course there is the point that many holes were found in the model as the implementation was built that would probably not have been found without building TP: This is a standard result of building an implementation program such as TP. Two lessons, however, stand out as primary:

- Primary Lesson 1: The majority of multi-bug tutoring situations are not one-off, distinct situations. Rather, most multi-bug tutoring situations are built up of known groups of multiple bugs that are commonly seen. In the Rainfall Task, for example, common patterns are missing output guards; spurious or malformed initializations; incorrect updates of accumulation variables, etc.

This fact became especially clear as TP was constructed. Because it was necessary to represent the knowledge that tutors use to generate tutorial plans for multiple bugs, an economical representation was desirable. The representations that were developed for TP are based on common patterns of multiple bugs.

- Primary Lesson 2: Tutorial planning for common multi-bug situations can be viewed as a pattern classification task in which the generation of a tutorial plan is based on identifying known patterns of bugs, selecting goals appropriate for the patterns of bugs, and selecting strategies that can achieve the goals.

Primary Lesson 2 is closely related to Primary Lesson 1. TP's representation of common multiple bug tutoring situations consists of a knowledge base of known patterns. For example, missing output guards form a known pattern that TP could find and then use to constrain the selection of tutorial goals and strategies. Thus, Primary Lesson 2 is that the generation of tutorial plans for common patterns of multiple bugs can be viewed as pattern-based.

5.9. Chapter Summary

This chapter has described TP, the implementation of the model of tutorial planning described in the previous three chapters. TP was described at two levels. First, TP was described at a general level, to give a high-level overview of its data structures and control flow. Second, a more detailed account of TP gave the specifics of its central data structures and the details of its processing mechanism. The final section presented two primary lessons that resulted from constructing the implementation. The two lessons revolved around the point that the generation of tutorial plans for common multiple bug tutoring situations can be viewed as a process driven by the recognition of known patterns of multiple bugs.

Chapter 6

STRENGTH OF CURRENT IMPLEMENTATION

6.1. Introduction

The previous chapters described the problem of tutoring a student who makes multiple bugs, the need for tutorial plans to guide the tutoring, the knowledge that human tutors use to generate tutorial plans, and TP, the computer program that is claimed to generate human-like tutorial plans.

The major question at this point is "How strong is the implementation of the tutorial planner?" That is, how much does TP know and how good are TP's tutorial plans? The purpose of this chapter is to present an evaluation of these two aspects of the strength of TP. The evaluation of TP therefore addresses two primary Evaluation Questions:

• Evaluation Question 1 (E.Q.1): How much knowledge does TP have about novice programming tasks and the bugs that students make when they try to solve those tasks?

• Evaluation Question 2 (E.Q.2): How good are the tutorial plans that TP generates?

This remainder of this chapter is divided into two main sections, each dealing with one of the two Evaluation Questions.

6.2. Q. 1: How Much TP Knows About Tasks and Bugs

6.2.1. Introduction

As the previous two chapters have shown, TP has four general categories of knowledge, each of which has several types. For example, TP's knowledge about the domain of programming includes the knowledge type of task specifications. TP's knowledge about bugs includes the knowledge type about where in the plan bugs are located. Because TP operates in the domain of novice Pascal programming, an important evaluation issue is the extent of TP's knowledge about task specifications and bugs. The purpose of this section, therefore, is to describe how much knowledge TP has about programming tasks and bugs.

6.2.2. Task Specifications TP Knows About

6.2.2.1. Introduction

TP has representations of the goals in the task specifications for two assignments, the Rainfall Task and the Reformatting Task. The task specifications are represented in a connected graph of goals, as described in Chapter 5: Each node in the graph identifies a goal, connects to parent and children goals, and identifies plans that can solve the goal.

6.2.2.2. The Rainfall Specification

The specification for the Rainfall Task consists of approximately 40 goals. The goals range from the very general goal of correctly understanding the task specification as described in the assignment, to including a main processing loop.

As Figure 6.1 shows, the goals of the Rainfall Task can be grouped into several primary categories. These categories are: to declare variables; to initialize variables; to update variables; to perform appropriate calculations, and to output the values of selected variables.

6.2.2.3. The Reformatting Specification

The specification for the Reformatting Task consists of approximately 90 goals. The Reformatting Task requires many more variables than the Rainfall Task, and nearly all variables require the same goals to be solved for them.

As Figures REFORMATGIOALS1 and REFORMATGIOALS2 show, there are approximately 10 variables which must be declared, initialized, repeatedly input, and guarded. Several calculations must be performed as intermediate steps to the final calculation. A subset of the variables must appear in documented output statements. For example, PROBLEMTYPE must be input, guarded, updated, reinput if necessary, and finally written out in a documented output statement.

Name of Goal	Description of Goal
UNDERSTAND-SPEC	Identify goals in specification correctly
DECLARE-SENTINEL	Declare the variable for the loop termination condition
DECLARE-AVERAGERAIN	Declare REAL variable for AVERAGERAIN
DECLARE-DAILYRAIN	Declare REAL variable for DAILYRAIN
DECLARE-RAINDAYS	Declare INTEGER counter for RAINDAYS
DECLARE-TOTALDAYS	Declare INTEGER counter for TOTALDAYS
DECLARE-MAXRAINFALL	Declare REAL variable for MAXRAINFALL
DECLARE-TOTALRAINFALL	Declare REAL variable for TOTALRAINFALL
INIT-DAILYRAIN	Initialize DAILYRAIN
INIT-RAINDAYS	Initialize RAINDAYS
INIT-TOTALDAYS	Initialize TOTALDAYS
INIT-MAXRAINFALL	Initialize MAXRAINFALL
INIT-TOTALRAINFALL	Initialize TOTALRAINFALL
INIT-SENTINEL	Initialize sentinel
CALC-AVERAGERAIN	Calculate AVERAGERAIN
UPDATE-DAILYRAIN	Update DAILYRAIN
UPDATE-RAINDAYS	Update RAINDAYS
UPDATE-TOTALDAYS	Update TOTALDAYS
UPDATE-MAXRAINFALL	Update MAXRAINFALL
UPDATE-TOTALRAINFALL	Update TOTALRAINFALL
GUARD-CALC-AVERAGERAIN	Guard calculation of AVERAGERAIN
GUARD-UPDATE-DAILYRAIN	Guard update of DAILYRAIN
GUARD-UPDATE-TOTALRAINFALL	Guard update of TOTALRAINFALL
GUARD-UPDATE-RAINDAYS	Guard update of RAINDAYS
GUARD-UPDATE-TOTALDAYS	Guard update of TOTALDAYS
GUARD-UPDATE-MAXRAINFALL	Guard update of MAXRAINFALL
GUARD-OUTPUT-AVERAGERAIN	Guard output of AVERAGERAIN
GUARD-OUTPUT-RAINDAYS	Guard output of RAINDAYS
GUARD-OUTPUT-TOTALDAYS	Guard output of TOTALDAYS
GUARD-OUTPUT-MAXRAINFALL	Guard output of MAXRAINFALL
GUARD-MAIN-LOOP	Guard main loop with sentinel
GUARD-RETRY	Guard retry on invalid input
OUTPUT-AVERAGERAIN	Output AVERAGERAIN
OUTPUT-RAINDAYS	Output RAINDAYS
OUTPUT-TOTALDAYS	Output TOTALDAYS
OUTPUT-MAXRAINFALL	Output MAXRAINFALL
CHECK-LOOP-CONDITION	Check loop termination condition
MAIN-LOOP	Implement main loop

Figure 6.1: Rainfall Task Specification Goals

Name of Goal	Description of Goal
UNDERSTAND-SPEC	Understand goals in specification correctly
DECLARE-ID	Declare ID INTEGER
DECLARE-STARTHOUR	Declare STARTHOUR INTEGER
DECLARE-STARTMINUTE	Declare STARTMINUTE INTEGER
DECLARE-STARTSECOND	Declare STARTSECOND INTEGER
DECLARE-ENDHOUR	Declare ENDHOUR INTEGER
DECLARE-ENDMINUTE	Declare ENDMINUTE INTEGER
DECLARE-ENDSECOND	Declare ENDSECOND INTEGER
DECLARE-STARTTIME	Declare STARTTIME REAL
DECLARE-ENDTIME	Declare ENDTIME REAL
DECLARE-ELAPSEDTIME	Declare ELAPSEDTIME REAL
DECLARE-PROBLEMTYPE	Declare PROBLEMTYPE CHAR
DECLARE-ACCURACY	Declare PROBLEMTYPE CHAR
DECLARE-ANSWER	Declare ANSWER CHAR
INIT-ID	Initialize ID
INIT-STARTHOUR	Initialize STARTHOUR
INIT-STARTMINUTE	Initialize STARTMINUTE
INIT-STARTSECOND	Initialize STARTSECOND
INIT-ENDHOUR	Initialize ENDHOUR
NIT-ENDMINUTE	Initialize ENDMINUTE
INIT-ENDSECOND	Initialize ENDSECOND
INIT-STARTTIME	Initialize STARTTIME
INIT-ENDTIME	Initialize ENDTIME
INIT-ELAPSEDTIME	Initialize ELAPSEDTIME
INIT-PROBLEMTYPE	Initialize PROBLEMTYPE
INIT-ACCURACY	Initialize ACCURACY
INIT-ANSWER	Initialize ANSWER
UPDATE-ID	Update ID
UPDATE-STARTHOUR	Update STARTHOUR
UPDATE-STARTMINUTE	Update STARTMINUTE
UPDATE-STARTSECOND	Update STARTSECOND
UPDATE-ENDHOUR	Update ENDHOUR
UPDATE-ENDMINUTE	Update ENDMINUTE
UPDATE-ENDSECOND	Update ENDSECOND
UPDATE-STARTTIME	Update STARTTIME
UPDATE-ENDTIME	Update ENDTIME
UPDATE-PROBLEMTYPE	Update PROBLEMTYPE
UPDATE-ACCURACY	Update ACCURACY
UPDATE-ANSWER	Update ANSWER
UPDATE-ALLDATA	Update all variables
PROMPT-UPDATE-ALLDATA	Prompt update of all variables
PROMPT-UPDATE-ID	Prompt update of ID
PROMPT-UPDATE-STARTHOUR	Prompt update of STARTHOUR
PROMPT-UPDATE-STARTMINUTE	Prompt update of STARTMINUTE
PROMPT-UPDATE-STARTSECOND	Prompt update of STARTSECOND

Figure 6.2: Reformatting Task Specification Goals

Name of Goal	Description of Goal
PROMPT-UPDATE-ENDHOUR	Prompt update of ENDHOUR
PROMPT-UPDATE-ENDMINUTE	Prompt update of ENDMINUTE
PROMPT-UPDATE-ENDSECOND	Prompt update of ENDSECOND
PROMPT-UPDATE-STARTTIME	Prompt update of STARTTIME
PROMPT-UPDATE-ENDTIME	Prompt update of ENDTIME
PROMPT-UPDATE-PROBLEMTYPE	Prompt update of PROBLEMTYPE
PROMPT-UPDATE-ACCURACY	Prompt update of ACCURACY
PROMPT-UPDATE-ANSWER	Prompt update of ANSWER
GUARD-UPDATE-ID	Guard update of ID
GUARD-UPDATE-STARTHOUR	Guard update of STARTHOUR
GUARD-UPDATE-STARTMINUTE	Guard update of STARTMINUTE
GUARD-UPDATE-STARTSECOND	Guard update of STARTSECOND
GUARD-UPDATE-ENDHOUR	Guard update of ENDHOUR
GUARD-UPDATE-ENDMINUTE	Guard update of ENDMINUTE
GUARD-UPDATE-ENDSECOND	Guard update of ENDSECOND
GUARD-UPDATE-STARTTIME	Guard update of STARTTIME
GUARD-UPDATE-ENDTIME	Guard update of ENDTIME
GUARD-UPDATE-PROBLEMTYPE	Guard update of PROBLEMTYPE
GUARD-UPDATE-ACCURACY	Guard update of ACCURACY
GUARD-UPDATE-ANSWER	Guard update of ANSWER
GUARD-UPDATE-ALLDATA	Guard update of all variables
RETRY-ID	Reinput ID on invalid value
RETRY-STARTHOUR	Reinput STARTHOUR on invalid value
RETRY-STARTMINUTE	Reinput STARTMINUTE on invalid value
RETRY-STARTSECOND	Reinput STARTSECOND on invalid value
RETRY-ENDHOUR	Reinput ENDHOUR on invalid value
RETRY-ENDMINUTE	Reinput ENDMINUTE on invalid value
RETRY-ENDSECOND	Reinput ENDSECOND on invalid value
RETRY-PROBLEMTYPE	Reinput PROBLEMTYPE on invalid value
RETRY-ACCURACY	Reinput ACCURACY on invalid value
RETRY-ALLDATA	Reinput all variables on invalid values
CALC-STARTTIME	Calculate STARTTIME
CALC-ENDTIME	Calculate ENDTIME
CALC-ELAPSEDTIME	Calculate ELAPSEDTIME
GUARD-CALC-ELAPSETIME	Guard calculation of ELAPSEDTIME
MAIN-LOOP	Use loop to repeat
OUTPUT-ID	Output ID
OUTPUT-PROBLEMTYPE	Output PROBLEMTYPE
OUTPUT-ELAPSEDTIME	Output ELAPSEDTIME
OUTPUT-ACCURACY	Output ACCURACY
DOCUMENT-OUTPUT-ID	Document the output of ID
DOCUMENT-OUTPUT-ACCURACY	Document the output of ACCURACY
DOCUMENT-OUTPUT-PROBLEMTYPE	Document the output of PROBLEMTYPE
DOCUMENT-OUTPUT-ELAPSEDTIME	Document the output of ELAPSEDTIME

Figure 6.3: More Reformatting Task Specification Goals

6.2.3. TP's Knowledge About Bugs

6.2.3.1. Introduction

TP's knowledge of bugs is currently stored in a static database that contains approximately 114 bugs. The database is partitioned into 69 Rainfall Task bugs and 45 Reformatting Task bugs. Each bug in the database is represented in a form that could be generated by a program such as PROUST (Johnson, 1985) or CHIRON (Sack, 1988). The database of bugs is simply a convenience for developing the tutorial planner and it is intended that, in a full tutoring system for programming, bugs will be detected and represented dynamically for each student's program. Thus, TP is not inherently dependent on a "bug database" as are e. g., MENO-II (Soloway, Woolf, Barth, & Rubin, 1981) or BRIDGE (ref).

Bugs in the database can be combined as desired to construct many different programs. The remainder of this section discusses the Rainfall bugs and the Reformatting bugs that are currently in TP's database and the approximate percentages of student bugs that they account for.

6.2.3.2. Rainfall Bugs

The 70 Rainfall bugs in TP's database are a compilation of bugs from the programs in Bug Catalogue I (Johnson, et al. 1983) and additional

bugs from programs generated by students in several introductory Pascal courses. The 70 bugs account for more than 60% of the bugs that students make when they try to solve the Rainfall Task (Johnson, et al., 1983).

Figures 6.4 and 6.5 list all the Rainfall Task bugs in the database and give a brief explanation of each.

6.2.3.3. Reformatting Bugs

The 45 Reformatting bugs in TP's database were selected form programs that appear in Bug Catalogue III (Spohrer, et al., 1985). The 45 Reformatting bugs account for more than 60% of the bugs that students make when they write programs for the Reformatting Task (Spohrer, et al., 1985).

Figure 6.6 lists all the Reformat Task bugs and describes each.

Name of Bug	Description of Bug
INIT-DAILYRAIN-CLOBBER-READ-DAILYRAIN	Spurious init destroys first value of DAILYRAIN
MISSING-INIT-RAINDAYS	Counter for RAINDAYS not initialized
NO-GUARD-RAINDAYS-COUNTER	Counter for RAINDAYS incremented unconditionally
NO-READ-DAILYRAIN-IN-LOOP	Input for DAILYRAIN missing from loop
UPDATE-TOTALRAINFALL-BELOW-LOOP	TOTALRAINFALL not accumulated in loop
DAILYRAIN-OUTPUT-FOR-RAINDAYS	Wrong variable output for RAINDAYS
NO-GUARD-OUTPUT-AVERAGERAIN	AVERAGERAIN output regardless of input
NO-GUARD-OUTPUT-MAXRAINFALL	MAXRAIFALL output regardless of input
WRONG-TYPE-FOR-AVERAGERAIN	AVERAGERAIN declared as an INTEGER
WRONG-OPERATOR-AVERAGERAIN-CALC	INTEGER divide used, not REAL divide
MISSING-INIT-DAILYRAIN-NEQ-SENTINEL	DAILYRAIN not guaranteed unequal to sentinel on 1st loop pass
MISSING-GUARD-DAILYRAIN-EQUAL-SENTINEL	No guard against DAILYRAIN equal to loop termination value
SENTINEL-ADD-IN-TOTALRAINFALL	Sentinel value accumulated into TOTALRAIN
RUNNING-TOTAL-TOTALRAIN-AS-COUNTER	TOTALRAIN incremented not accumulated
BACKOUT-SENTINEL-AVERAGERAIN-CALC	Sentinel value subtracted from TOTALRAINFALL
SPURIOUS-INITIALIZATION-AVERAGERAIN	AVERAGERAIN unnecessarily initialized
WRONG-SENTINEL-TEST-VALUE	Loop termination condition incorrect
MALFORMED-UPDATE-TOTALRAINFALL	TOTALRAIN not accumulated correctly
MALFORMED-UPDATE-TOTALDAYS	TOTALDAYS not accumulated correctly
MALFORMED-UPDATE-MAXRAINFALL	MAXRAINFALL not accumulated correctly
MALFORMED-UPDATE-RAINDAYS	RAINDAYS not accumulated correctly
AVERAGERAIN-CALC-IN-LOOP	AVERAGERAIN calculated each pass of loop
WRONG-TYPE-FOR-TOTALDAYS	TOTALDAYS declared REAL
TOTALDAYS-INITIALIZED-TO-1-NOT-0	Wrong initial value for TOTALDAYS
LOOP-READ-CLOBBER-INITIAL-DAILYRAIN	DAILYRAIN read above loop and in loop before updates
MISSING-INIT-TOTALDAYS	TOTALDAYS not initialized
MISSING-INIT-TOTALRAINFALL	TOTALRAINFALL not initialized
MISSING-INIT-MAXRAINFALL	MAXRAINFALL not initialized
MISPLACED-BEGIN-AT-TOP-LOOP	BEGIN for loop block misplaced
NULL-REPEAT-UNTIL-LOOP	Empty REPEAT/UNTIL loop after main loop
WRONG-LOOP-GUARD-BINOP	Wrong test in loop guard
MISPLACED-AVERAGERAIN-CALC	Calculation of AVERAGERAIN not after loop, before output
RAINDAYS-UPDATE-FOR-TOTALDAYS	RAINDAYS performs function of TOTALDAYS
WRONG-DIVISOR-AVERAGERAIN	TOTALDAYS not divisor in AVERAGERAIN calculation

Figure 6.4: Rainfall Task Bugs

Name of Bug	Description of Bug
WRONG-TYPE-FOR-MAXRAINFALL	MAXRAINFALL declared as INTEGER
WRONG-TYPE-FOR-TOTALRAINFALL	TOTALDAYS declared as REAL
WRONG-TYPE-FOR-DAILYRAIN	DAILYRAIN declared INTEGER
MISSING-MAIN-BEGIN/END-IN-LOOP	No BEGIN/END block in main loop
SPURIOUS-GUARD-UPDATE-TOTALDAYS	TOTALDAYS unnecessarily guarded
SPURIOUS-GUARD-UPDATE-TOTALRAINFALL	TOTALRAINFALL unnecessarily guarded
SPURIOUS-GUARD-UPDATE-AVERAGERAIN	AVERAGERAIN unnecessarily guarded
MISSING-OUTPUT-AVERAGERAIN	AVERAGERAIN not output
MISSING-OUTPUT-MAXRAINFALL	MAXRAINFALL not output
MISSING-OUTPUT-TOTALDAYS	TOTALDAYS not output
MISSING-OUTPUT-RAINDAYS	RAINDAYS not output
MISSING-UPDATE-MAXRAINFALL	MAXRAINFALL not updated
SPURIOUS-COUNTER-DRYDAYS	Number of dry days unnecessarily counted
MALFORMED-BOOLEAN-MAIN-LOOP-GUARD	Loop termination test wrong
MISSING-INITIAL-VALUE-DAILYRAIN	DAILYRAIN not given initial value
MISSING-UPDATE-TOTALDAYS	TOTALDAYS not counted
SPURIOUS-ASSIGNMENT-MAXRAINFALL	MAXRAINFALL assigned regardless of value of DAILYRAIN
MISSING-GOAL-OUTPUT-TOTALDAYS	TOTAOLDAYS not output
MISPLACED-OUTPUT-RAINDAYS	RAINDAYS not output after main loop
MISPLACED-OUTPUT-MAXRAINFALL	MAXRAINFALL not output after main loop
WRONG-CONSTANT-IN-LOOP-GUARD	Incorrect value for sentinel
SPURIOUS-READ-DAILYRAIN	DAILYRAIN unnecessarily guarded
WRONG-VARIABLE-OUTPUT-FOR-TOTALDAYS	TOTALDAYS not output for total number of days
RAINDAYS-INITIALIZED-TO-1-NOT-0	RAINDAYS initialized to 1 instead of 0
MISSING-SENTINEL-GUARD-AFTER-RETRY	Sentinel not detected after re-input
MISPLACED-INIT-RAINDAYS	RAINDAYS initialized in wrong place
MISPLACED-INIT-COUNTER-X-RAIN	Spurious variable initialized in wrong place
SPURIOUS-OUTPUT-TOTALDAYS	TOTALDAYS output too often
SPURIOUS-OUTPUT-TOTALRAIN	TOTALRAIN output too often
SPURIOUS-OUTPUT-AVERAGERAIN	AVERAGERAIN output too often
MISPLACED-INIT-MAXRAINFALL	Initialization of MAXRAINFALL misplaced
SPURIOUS-OUTPUT-MAXRAINFALL	MAXRAINFALL output too often
SPURIOUS-OUTPUT-MAXDAYS	Spurious variable output too often
WHILE-FOR-IF-IN-MAXRAINFALL-UPDATE	Update of MAXRAINFALL in WHILE loop

Figure 6.5: More Rainfall Task Bugs

Name of Bug	Description of Bug
MISSING-BEGIN/END-FOR-MAIN-LOOP	No BEGIN/END for main loop
BLANK-CHAR-READ-FOR-ACCURACY	READLN reads blank for ACCURACY
WRONG-BOOLEAN-OP-OR-FOR-AND	OR for AND in Boolean test
MISS-GUARD-STARTTIME-GT-ENDTIME	STARTTIME > ENDTIME undetected
MISS-PLAN-STARTTIME-GT-ENDTIME	Calculation doesn't handle STARTTIME > ENDTIME
MISS-PROMPT-FOR-LOOP-CONTINUE	No prompt to continue
MISS-PROMPT-FOR-NEW-DATA	No prompt to enter more data
MISS-REINPUT-OF-DATA	No reinput of data
MISS-BEGIN/END-PAIR-FOR-BLOCK	Missing BEGIN/END
FORCE-INPUT-ALL-DATA-ON-RETRY	Input error forces all variables to be reinput
WRONG-BOOLEAN-OP-AND-FOR-OR	AND for OR in Boolean test
OFF-BY-ONE-GUARD-STARTHOUR	STARTHOUR could be 13
SPURIOUS-BEGIN/END-PAIR	Unnecessary BEGIN/END
MALFORMED-GUARD-BOOLEAN-OP-PRECPT	Wrong test for PROBLEMTYPE
MALFORMED-GUARD-BOOLEAN-OP-PRECAC	Wrong test for ACCURACY
MALFORMED-GUARD-BOOLEAN-OPERATCID	Wrong test for ID
MISS-GUARD-FOR-VALIDITY-OF-TIME-DATASH	No guard for invalid STARTHOUR
MISS-GUARD-VALIDITY-OF-TIME-DATASM	No guard for STARTMINUTE
MISS-GUARD-VALIDITY-OF-TIME-DATASS	No guard for STARTSECOND
MISS-GUARD-VALIDITY-OF-TIME-DATAEH	No guard for ENDHOUR
MISS-GUARD-VALIDITY-OF-TIME-DATAEM	No guard for ENDMINUTE
MISS-GUARD-VALIDITY-OF-TIME-DATAES	No guard ELAPSEDTIME
WRONG-PLAN-GUARD-START-TIME-GT-END-TIME	STARTTIME > ENDTIME Calc wrong
MISS-RETRY-ON-BAD-DATA	No reinput on invalid data
MISS-PROMPT-FOR-NEGATIVE-ID-TO-STOP	No prompt to enter negative for ID to stop
STARTTIME-AND-ENDTIME-REVERSED-CALC	STARTTIME and ENDTIME reversed in calculation
SPURIOUS-VAR-READ-PROBLEMTYPE	PROBLEMTYPE read too often
SPURIOUS-VAR-READ-ID-OUTPUT	ID read too often
SPURIOUS-VAR-READ-PROBLEMTYPE-OUTPUT	PROBLEMTYPE read before write
SPURIOUS-VAR-READ-ENDTIME-OUTPUT	ENDTIME read before write
SPURIOUS-VAR-READ-ELAPSEDTIME-OUTPUT	ELAPSEDTIME read before write
PURIOUS-VAR-READ-ACCURACY-OUTPUT	ACCURACY read before write
MISS-GUARD-FOR-STARTHOUR-GT-0	No guard for STARTHOUR > 0
OFF-BY-ONE-GUARD-STARTMINUTE-59	STARTMINUTE could be 60
OFF-BY-ONE-GUARD-STARTSECOND-59	STARTSECOND could be 60
OFF-BY-ONE-GUARD-ENDHOUR-12	ENDHOUR could be 13
OFF-BY-ONE-GUARD-ENDMINUTE-59	ENDMINUTE could be 60
OFF-BY-ONE-GUARD-ENDSECOND-59	ENDSECOND could be 60
MISPLACED-REINPUT-ID	Reinput on invalid misplaced
WRONG-PLAN-CALC-ELAPSEDTIME	ELAPSEDTIME not calculated correctly
NO-DESCRIPTION-OF-OUTPUT-ID	Output of ID not documented
NO-DESCRIPTION-OF-OUTPUT-AC	Output of ACCURACY not documented
NO-DESCRIPTION-OF-OUTPUT-PT	Output of PROBLEMTYPE not documented
NO-DESCRIPTION-OF-OUTPUT-ET	Output of ELAPSEDTIME not documented

Figure 6.6: Reformat Task Bugs

6.2.4. Summary: E. Q. 1

TP has representations for two task specifications and the the most common bugs that students make when they try to solve the tasks (Spohrer, et al., 1985). TP represents each task as a conjunction of goals. There are approximately 40 goals for the Rainfall Task and 90 goals for the Reformatting Task.

The bugs in TP's knowledge base are associated with specific goals in the task representations. The bugs in TP's database account for approximately 60 percent of the bugs that students make when they try to solve the Rainfall Task and the Reformatting Task.

Many of the task goals, and many of the bug types, are common across the two task specifications. This communality leads to a degree of domain generality, a topic that is discussed in Chapter 8.

6.3. Q. 2: How Good TP's Tutorial Plans Are

6.3.1. Introduction

The empirical validation of the tutorial plans that TP can produce was designed to determine whether TP's tutorial plans were as good as the tutorial plans generated by experienced human tutors. Two studies were performed that presented experienced human tutors with tutorial plans generated by TP. The human tutors rated various features of TP's tutorial plans. For example, the tutors were asked

to rate the acceptability of each plan, how willing they would be to use the plan to tutor the buggy program for which the tutorial plan was generated, and so forth. As well, the tutors were given ample opportunity to identify aspects of each tutorial plan that they did not like and were able to indicate how they would modify undesirable aspects of the tutorial plan.

In the first study, experienced tutors were asked to criticize TP's best tutorial plan for each of five buggy programs. In the second study, experienced tutors were presented with three tutorial plans for each of five buggy programs. TP generated one of the tutorial plans and the others were generated by the experimenter. The tutor's task was to rate each of the tutorial plans according to several features and then to rank order the three tutorial plans according to their goodness.

The argument that the first empirical study attempts to support can be stated as follows:

- Experienced human tutors generate, and use, good tutorial plans.
- Experienced tutors can identify the aspects of a tutorial plan that are good and the aspects the tutorial plan that are bad.
- Tutors can say how the bad aspects of the plan could be modified to make it better.
- If the agreement among several tutors about the acceptability TP's tutorial plans is statistically significant, and they agree that the tutorial plans are acceptable, then the study provides strong prima facie evidence that TP's tutorial plans are similar to the tutorial plans that the human tutors would use.

The argument that the second empirical study attempts to support is the following:

- Experienced tutors can recognize good tutorial plans and bad tutorial plans for the same buggy programs.
- Experienced tutors can identify how good and how bad the tutorial plans are according to how the plans address the five tutorial planning decisions.
- If tutors agree about which tutorial plans are best, and which are bad, and the bad tutorial plans were generated by ignoring one or more of the five tutorial planning decisions, then the study provides evidence that the five tutorial planning decisions are key to generating good tutorial plans.

Thus, the validation studies for the quality of TP's tutorial plans are based on extensive evaluation by experienced human tutors.

6.3.2. Description of Empirical Study 1

6.3.2.1. Subjects

The four subjects in the empirical test of the acceptability of TP's tutorial plans were all graduate students in the Computer Science Department at Yale University. Each subject was an experienced tutor who had tutored students in novice Pascal courses (among others) for at least two terms. Because of the amount of data collected from each subject and the amount of time required to complete the task (the average was more than six hours), four

subjects participated in the study. Each subject produced approximately 90 pages of data. Only a small part of the data produced by each subject was analyzed for the validation study.

6.3.2.2. Materials

Subjects were asked to evaluate tutorial plans for 5 buggy Rainfall Task programs and 5 buggy Reformatting Task programs. There was an average of eight bugs per program. In each of the 10 stimulus programs the bugs were identified for the tutors because the goal was to evaluate tutorial plans, not bug finding. The ten programs were not in the set of programs from which TP's knowledge base was generated.

Figure 6.7 shows Program BMT085, one of the 5 buggy Rainfall programs: Program BMT085 has 12 bugs. Figure 6.8 shows the tutorial plan for the program shown in Figure 6.7. The bugs in the tutorial plan are divided into groups, objectives are identified for them, strategies are indicated for the objectives, a note is made about how important it is to tutor the bugs in the group and the groups of bugs are ordered. In this case, TP said that all but the final group of bugs, Group 5, definitely should be tutored. In the case of Group 5, TP said that the bugs should just be pointed out to the student.

The experienced tutors who served as subjects were asked to rate, on 7-point scales, several aspects of each tutorial plan. The leftmost point on the scale, corresponding to 1, represented the least desirable value on the scale. The rightmost point,

corresponding to 7, represented the most desirable value on the scale. Figure 6.9 shows the bulk of the scaled questions that the subjects were asked about each of the 10 buggy programs in the Validation Study.[2] The first question asks for a scaled global evaluation of the quality of the tutorial plan. As the additional questions in the figure show, tutors were asked to evaluate the grouping, ordering, and strategy selections for each tutorial plan. In addition, tutors were asked for global scaled evaluations of the intelligence of the student who wrote the program and how hard they thought it would be to tutor the student. As well, a final, open ended question gave subjects the opportunity to bring up any issues that they felt had not been covered adequately in the specific questions.

[2]These questions were spread out over four pages in the questionnaires the tutors filled out. The questions are minified to show them all in one figure.

```
/* BMT085 */
PROGRAM Noah (INPUT,OUTPUT);
    CONST
        Sentinel = 99999;
    VAR
BUG 1: WRONG TYPE DAILY RAINFALL
BUG 2: WRONG TYPE MAXIMUM
BUG 3: WRONG TYPE TOTAL RAINFALL
        DailyRainfall,  MaxRainfall, TempRaindays
        Raindays, TotalRainfall, TotalDays :  INTEGER;
        AverageRain : REAL
    BEGIN
BUG 4: NO INITIALIZATION TOTAL DAYS
        AverageRain := 0;
        MaxRainfall := 0;
        Raindays := 0;
        TotalRainfall := 0;
BUG 5: NO INITIALIZATION DAILY RAINFALL <> SENTINEL
BUG 6: MALFORMED LOOP GUARD (SHOULD BE <>)
        WHILE DailyRainfall <  99999 DO
            BEGIN
                WRITLEN('Enter rainfall data. ');
                READLN;
                READ(DailyRainfall);
                BackOutInvalid := BackoutInvalid + 1;
                WHILE DailyRainfall < 0 DO
                    BEGIN
                        WRITLEN('Enter rainfall value > 0. ');
                        READLN;
                        READ(DAILYRAINFALL);
                    END;
BUG 7: NO SENTINEL GUARD
                TotalRainfall := TotalRainfall + DailyRainfall;
                IF DailyRainfall = 0 THEN TempRainDays := 0;
                IF DailyRainfall > 0 THEN TempRainDays := 1;
                RainDays := RainDays + TempRainDays;
                IF DailyRainfall > MaxRainfall THEN
                    MaxRainfall := DailyRainfall;
BUG 8: AVERAGE CALCULATION IN LOOP
BUG 9: NO GUARD AGAINST RAINDAYS = 0
BUG 10: WRONG DIVISOR FOR AVERAGE RAIN
                AverageRain := TotalRainfall/RainDays;
            END;
BUG 11: NO GUARD AVERAGE OUTPUT
BUG 12: NO GUARD MAXIMUM OUTPUT
            WRITELN('There were ', RainDays :0:2 'rainy days in the period.');
            WRITELN('The average daily rainfall was ', AverageRain :0:2 'inches per day. ');
            WRITLEN('The highest daily rainfall was ' MaxRainfall :0:2, ' inches.');
        END.
```

Figure 6.7: One of 10 Buggy Programs in Evaluation Study

GROUP 1:

WHETHER TO TUTOR: DEFINITELY

Justification: Student may not realize that rainfall may not be integer value

BUGS IN GROUP:

Bug 1: Wrong type for Daily Rainfall
Bug 2: Wrong type for Maximum
Bug 3: Wrong type for Total Rainfall
Justification: All are problems with variable typing

ORDER OF BUGS: Doesn't matter

PLAN

1. Show student that there could be non-integer values of each variable.

GROUP 2:

WHETHER TO TUTOR: DEFINITLEY

Justification: Program will not produce correct result

BUGS IN GROUP:

Bug 10: Wrong divisor for Average Rain

ORDER OF BUGS:

PLAN

1. Ask student why s/he used Rainy days for the divisor

GROUP 3:

WHETHER TO TUTOR: DEFINITELY

Justification: common pattern to watch out for

BUGS IN GROUP:

Bug 5: No initialization of rainfall not equal to sentinel
Bug 6: Malformed loop guard
Bug 7: No guard rainfall equal to sentinel
Justification: Common pattern in this type of loop

ORDER OF BUGS:

Bug 5: No initialization of rainfall not equal to sentinel
Bug 7: No guard rainfall equal to sentinel
Bug 6: Malformed loop guard

PLAN

1. Simulate loop, especially with boundary case of DailyRainfall equal to sentinel. This will get student to articulate problem. Then hint at use of guard. Then "point out" that the condition is not quite righ

GROUP 4:

WHETHER TO TUTOR: DEFINITELY

Justification: Not the standard way of doing things

BUGS IN GROUP:

Bug 4: Average calc in loop

ORDER OF BUGS:

PLAN

1. Ask student why the average calc is in loop.

GROUP 5:

WHETHER TO TUTOR: IF TIME REMAINS

Justification: These bugs are not crucial.

BUGS IN GROUP:

Bug 9: No guard for Raindays = 0
Bug 11: No Guard for output of Average
Bug 12: No guard for output of Maximum

ORDER OF BUGS:

Probably start with Bug 9 because it could cause the program to blow up.

PLAN

1. These bugs should just be pointed out

Figure 6.8: A Tutorial Plan for BMT085 in Evaluation Study

6.3.2.3. Procedure

Subjects were given all the stimulus materials (approximately 150 pages) at one time. They were allowed to work at their own pace with the following provisos:

- Subjects were to finish all five Rainfall Task programs at one sitting.
- Subjects were to finish all five Reformatting Task programs at one sitting.
- Subjects were to complete the task in two days or less.

Subjects worked in their offices, or at home. Each subject took at least six hours to complete the task. Some tutors took over seven hours. Subjects said that the task was tedious but that they believed that their responses were realistic. Subjects were paid for their participation.

6.3.2.4. Results

The results of the validation study demonstrated that the tutorial plans evaluated by the tutors were more than acceptable to them. Figure 6.10 shows the results for the seven main rating scale questions asked of the tutors shown in Figure 6.9.

According to common procedure for scaled data, marks on the scale line were converted to, and analyzed as, decimal numbers. The three columns corresponding to each question in Figure 6.10 are averages across all 10 programs (five Rainfall programs and five

Reformatting programs) and all four tutors. Thus, each entry represents 40 observations.

The entries in the column corresponding to the mean of all the tutors' judgements shows that, on all seven scales, the tutorial plans generated by TP were more than acceptable. Note especially that TP's grouping decisions and strategy selection decisions were particularly good, according to the human tutors. The standard deviations, as well, were smaller than might be expected if different tutors had radically different ideas about the attributes that constitute a good tutorial plan. The last column identifies the conclusions that can be drawn from the validation study. As the conclusions column for the first five questions shows, tutors believed that TP generates tutorial plans that are more than acceptable.

The final question in the Figure 6.10 provides an intriguing insight into the tutors attitudes toward TP's tutorial plans: The tutors in the validation study said that they would be willing to use TP's plans to tutor the student who wrote the buggy program corresponding to the plan. This can be interpreted to mean that they genuinely felt TP's plans were good.

QUESTION	MEAN	STANDARD DEV.	CONCLUSION
Global Quality	5.0	2.0	+
Grouping	5.9	1.4	++
Tutor or Not	6.0	1.7	++
Group Ordering	5.2	1.7	+
Strategies	6.7	0.5	++
Would Use?	5.4	1.5	+

(N=40 per Question)

Figure 6.10: Results of Scaled Questions in the Validation Study

6.3.3. Description of Empirical Study 2

6.3.3.1. Subjects

10 subjects participated in the second empirical test of the acceptability of TP's tutorial plans. Subjects were either graduate students, professional staff, or faculty in the Computer Science Department at Yale University who had not participated in Empirical Study 1. Each subject was an experienced tutor who had tutored students in novice Pascal courses (among others) for at least two terms. Each subject produced approximately 20 pages of data.

6.3.3.2. Materials

Subjects evaluated three different tutorial plans for each of five buggy programs, for a total of 15 tutorial plans. The three tutorial plans for each buggy program were designed to be more, or less, like those that experienced human tutors generate:

- One tutorial plan was generated by TP. This tutorial plan was generated by making the five Tutorial Planning Decisions and therefore the plan differed for each of the five buggy programs.
- One tutorial plan used none of the knowledge and rules that TP used. This tutorial plan was exactly the same for each of the five buggy programs. The five tutorial decisions were made as follows:
    - Tutorial Decision 1: All bugs are tutored.
    - Tutorial Decision 2: Bugs are tutored separately.
    - Tutorial Decision 3: Bugs are tutored in the order that they appear in the program.
    - Tutorial Decision 4: The only tutorial goal is to make the program work.
    - Tutorial Decision 5: The tutorial strategy is to point out the bug and *tell* the student how to fix it.
- One tutorial plan used some of the knowledge and rules that TP used. Specifically:
    - Tutorial Decision 1: All bugs are tutored.

- Tutorial Decision 2: Bugs are tutored separately.
- Tutorial Decision 3: Bugs are tutored in order of importance starting with the most important bugs.
- Tutorial Decision 4: Tutorial goals are as generated by TP.
- Tutorial Decision 5: The tutorial strategy is to point out the bug and *ask* the student how to fix it.

For each tutorial plan, subjects were asked to indicate the extent to which they agreed with seven statements about the tutorial plan. Five of the seven statements asked how much they agreed with the five Tutorial Planning Decisions. One statement asked subjects how good the tutorial plan was overall and the final statement asked how willing subjects would be to use the tutorial plan to help the student who wrote the buggy program.

Finally, subjects were required to rank order the three tutorial plans in terms of overall quality. No ties were permitted in the rankings. After the ranking task, subjects were given the opportunity to respond to some open-ended questions that allowed them to say anything they cared to about the study, the programs, the tutorial plans, and the task.

Q1.	*Overall*, this tutorial plan is as good as best possible
Q2.	Decision about *which bugs to tutor* is as good as best possible
Q3.	Decision about *bugs to tutor together* is as good as best possible
Q4.	Decision about *ordering of bugs* is as good as best possible
Q5.	Decision about *goals to achieve* is as good as best possible
Q6.	Decision about *strategies to use* is as good as best possible
Q7.	I would be willing to use this tutorial plan

Figure 6.11: Seven Statements about Tutorial Plans

6.3.3.3. Procedure

Subjects received all the stimulus materials in a single packet. They were allowed to work on the materials at their own pace. Subjects worked in their offices or at home. The task required much less time to complete than in Experiment 1. No subject needed more than 2 hours to complete the task.

6.3.3.4. Results

The results of Empirical Study 2 provide strong support for the claim that TP's tutorial plans are more better than tutorial plans generated without all of TP's knowledge and rules. The two main results of Empirical Study 2 show:

- First, when subjects are asked to rank the three tutorial plans for a multi-bug program by identifying the best tutorial plan, the worst tutorial plan, and the intermediate quality tutorial plan, TP's tutorial plans almost uniformly ranked ahead of the other two plans. In 38 out of a possible 40 cases, TP's tutorial plans were ranked first. The results of the tutorial plan ranking task are significant by sign test at beyond the .001 level of statistical significance.

- Second, when subjects are asked to indicate how strongly they agree with each of the seven positive statements about the three tutorial plans for each of the five multi-bug programs, they express much stronger agreement with the statements when they

refer to TP's tutorial plans than when they refer to plans generated without TP's knowledge and rules.

Figure 6.12 shows the seven statements and the number of times subjects agreed with the statements most strongly when they referred to TP's tutorial plans. The figure shows that, overwhelmingly, the tutors agreed that TP's tutorial plans were better than the other two in terms of 1) overall quality, 2) three of the five Tutorial Planning Decisions, and 3) how willing they would be to use the tutorial plan generated by TP.

Tutors did not rate TP's ordering and goal selection decisions as highly as the other three Tutorial Planning Decisions. There are several possible explanations for this. First, tutors may simply disagree more among themselves on these two Tutorial Planning Decisions than the other three. Second, TP may be deficient in making these two Tutorial Planning Decisions in comparison with the other three and tutors agree that TP is deficient i.e., it is not simply a a matter of an honest difference of opinion. Third, tutors may have been less certain of their own ordering and goal selection decisions and therefore were reluctant to rate any such decisions, including TP's, higher than others.

Regardless of which of these explanations turns out to be correct, it is *not* valid to conclude from the data that tutors thought that TP did *badly* on the goal and ordering decisions. To see this, notice that the proportions reported in Figure 6.12 are very conservative because, if two tutorial plans were *equally* acceptable for one of the seven questions, this was counted as a *loss* by TP's tutorial plan. In addition, out of the 280 responses (8 subjects X 5

programs X 7 questions) there were only 13 outright losses by the tutorial plans generated by TP. There were 51 ties which came almost exclusively from the tutorial plans for which the tutorial decisions for the intermediate plan were the same as TP's decisions. Thus, it is as if the glass were half full as opposed to half empty: Under a very conservative interpretation, experienced tutors preferred TP's decisions at least half the time.

	TP's Plan Rated Best
Q1. *Overall*	33/40
Q2. *which bugs to tutor*	34/40
Q3. *bugs to tutor together*	36/40
Q4. *ordering of bugs*	20/40
Q5. *goals to achieve*	23/40
Q6. *strategies to use*	34/40
Q7. willing to use	36/40

Figure 6.12: Number of Times out of 40 TP's Plan Rated Best

6.3.4. Summary: E. Q. 2

In summary, TP's tutorial plans were evaluated by human tutors in two separate studies. In the first study, tutors reported on the acceptability of each of ten tutorial plans according to several attributes such as overall quality, grouping decisions, and strategy selection. The results show that the tutorial plans were acceptable on the main features, including grouping, ordering, strategy

selection, and the willingness of the tutors to use the tutorial plans to tutor the student who wrote the buggy programs.

In the second study, tutors rated 15 tutorial plans -- three each for the five buggy programs. The three tutorial plans were simple, intermediate in complexity, or generated by TP. The results of the analyses of the rating data show that TP's tutorial plans are overwhelmingly preferred over the other tutorial plans.

6.4. Chapter Summary

This chapter has described the chief aspects of the evaluation of TP and the tutorial plans that TP generates. The evaluation of TP was focused on two evaluation questions. The first evaluation question, E. Q. 1, asks how much TP knows. The second evaluation question, E. Q. 2 addresses the quality of the tutorial plans that TP generates.

Two aspects of TP's knowledge are key to its performance, namely its knowledge about task specifications and its knowledge about bugs. Both of these kinds of knowledge are extensive and fairly general in that slight modifications of them can produce other task specifications and other bugs.

The tutorial plans that TP generates are considered to be more than acceptable by human tutors. The tutors' judgements about how well TP answered the five Tutorial Planning Questions showed that, at the least, TP's knowledge about how to generate tutorial plans is powerful enough to generate tutorial plans similar to the ones that

human tutors generate. Two studies addressing the acceptability of TP's tutorial plans supported the same conclusions.

Chapter 7

EXECUTING TUTORIAL PLANS & RECOVERING FROM FAILURE

7.1. Introduction

Once the tutor, whether human or computer, generates a tutorial plan, two issues arise. First, the tutorial plan must be executed --- the student does not receive any help until the tutorial plan is executed. Second, during the execution of the tutorial plan there are likely to be tutoring failures. The mechanism that executes the tutorial plan must be able to recover dynamically from failures: It is unreasonable for the tutor to try to anticipate, for every action of the tutorial plan, each possible reaction of the student and to generate a tutorial response for each possible reaction (Peachy & McCalla, 1986).

For example, if as in Figure 1.1, the student makes a bug because of a misconception about the scoping of a WHILE loop, the tutorial plan might specify that the tutor ask the student to perform a hand simulation of the main loop in order to lead the student to see how the loop behaves when it executes. Suppose that, when the tutor makes the request for the hand simulation, the student is unable to carry it out effectively; this is a common occurrence because students do not always know how to perform complete hand simulations (Spohrer, 1989). The tutorial plan thus fails at the point when the tutor asks the student to perform a hand simulation.

Any subsequent tutoring that depends on the results of the student's hand simulation of the loop will most likely be unsuccessful.

The problem for the tutorial plan execution mechanism is to recover from this failure. For example, one recovery procedure might be for the tutor to perform the hand simulation of the main loop while the student *observes* the results of the simulation. Because the tutor's goal is to lead the student to understand scoping of WHILE loops, and not specifically how to perform hand simulations of them, this is a reasonable recovery strategy.

The focus of this chapter is on tutorial plan execution and recovery from failures during tutorial plan execution. Although tutorial plan execution and recovery from failure are not the main topics of the dissertation they are important issues that must be addressed as the effort progresses to build a complete tutoring system for novice programming. The goal of this chapter is to present some descriptive analyses of the knowledge supporting tutorial plan execution and recovery from tutorial failure. The analyses of the empirical data bearing on these two issues are not as exhaustive as the analyses that led to the model of tutorial planning:

- In the case of tutorial plan execution, an attempt was made to provide a plausible breakdown into individual steps of the tutorial strategies described in Chapter 5.
- In the case of recovery from tutorial failure, interactions between tutors and students in the Failure Recovery Study were evaluated with the goal of identifying some common strategies that tutors use when their tutorial plans encounter minor failures e. g.,

the student does not understand how the data are stored in a file and the tutor draws a picture of the file with data in it.

7.2. Tutorial Plan Execution and Recovery from Failure

The work on tutorial plan execution and recovery resulted in a preliminary model of some of the knowledge required to execute tutorial plans and recover from failure. The model was implemented in a very simple program module called PEFR (Plan Execution and Failure Recovery). The implementation of PEFR is so simple that it will not be discussed further. However, it is useful to distinguish PEFR's functions from TP's functions

Figure 7.1 shows the primary functions of TP and PEFR. TP identifies appropriate tutorial strategies, such as asking the student to hand simulate a section of code with the objective of making the student aware of the existence of a bug. TP does not, however, specify in the tutorial plan the specific steps of the tutorial strategies. PEFR identifies the steps (such as identifying a section of code by describing it) to be taken for each strategy specified by TP and then prints a message corresponding to what the tutor would say if it were a complete tutoring system. No claim is made that humans identify the actions of tutorial plans at execution time. However, it seems unreasonable to suppose that humans specify all the actions at planning time.

Second, PEFR responds to tutorial failures by asking the human user of the program to serve as an oracle and to identify the cause

of the failure. PEFR then determines the action that is appropriate for the failure and prints a message corresponding to what the tutor would do in response to the failure e. g., "Identify effects of bug for student." This topic is considered later in this chapter. TP's main responsibility is thus the generation of the tutorial plan, which is passed to PEFR to be executed -- this is shown in Figure 7.2. PEFR's main responsibilities include executing the tutorial plan and recovering from tutorial failures.

It is important to note that these analyses are intended to be preliminary and to point the direction for future research. The tutorial plans generated by TP, and executed by PEFR, were not evaluated by trying them with students and observing their effects on the students' problem solving behavior: This type performance evaluation is intended as a future research topic. The tutorial plans generated by TP were evaluated in the study described in Chapter 6.

FUNCTIONS OF PROGRAM MODULE:

TP	PEFR
1. Make Tutorial Planning Decisions Group Order Tutor-Or-Not Objectives Strategies	1. Execute Tutorial Plan Expand Strategies into Steps Identify Actions for Steps Name Actions for Steps 2. Recover From Failure Ask Oracle for Cause of Failu Recovery Strategy Name Actions for Recovery

Figure 7.1: Functions of Tutorial Planner and Plan Executor

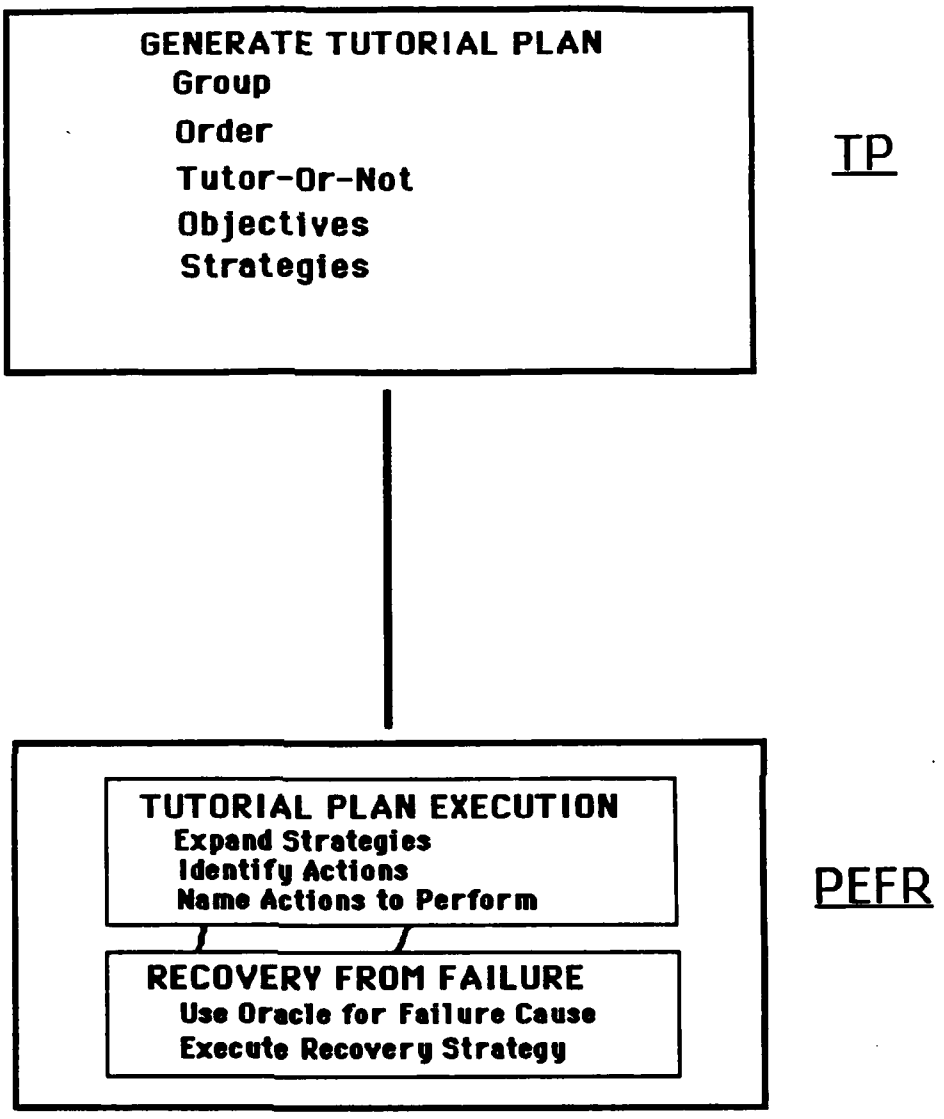


Figure 7.2: Functions of Tutorial Planner and Plan Executor

7.2.1. Tutorial Plan Execution

As noted above, TP specifies tutorial strategies for objectives but it is the responsibility of PEFR to expand each strategy selected by TP into its component steps.[3] The current work on tutorial plan execution attempted to address one issue in tutorial plan execution, namely the nature of the breakdown of tutorial strategies selected by TP into individual steps.

The breakdowns of the strategies into component steps presented in this chapter are based on the data from the Questionnaire Study and the Real Time Study. The analyses of the data were only deep enough to support what appear to be extremely *intuitive breakdowns of the strategies.* Tutors did not consistently specify the steps of their strategies in the Questionnaire Study and the data set from the Real Time Study are limited. Therefore, no strong claim is being made that the strategy breakdowns given here are the ones that all tutors use. Rather, the claim is made that these strategy breakdowns are intuitive, prototypical, and common, and suggest the direction for future research on the topic.

Figure 7.3 shows an example of the breakdown of one of the four strategies discussed in Chapter 5, "Simulate Values". The context in which TP specified this strategy is shown in Figure 3.2. In the program shown in Figure 3.2, the student has initialized the counter TOTALD. S to 1 to avoid a potential division by zero error

[3] PEFR simply looks up the strategy breakdowns in a table and prints them out. Because this is a trivial implementation, the program is not discussed further.

in the calculation of AVERAGERAIN. The result of the student's choice of initial values for TOTALDAYS is an undetected bug: TOTALDAYS will have an incorrect value (1) whenever the user of the program does not enter any valid values for RAINFALL. As part of the process of helping the student, the tutor specifies the objective of making the student aware of the bug --- the undesired side effect of the initialization. To achieve this objective the tutor uses the strategy of asking the student to simulate the program when the user does not enter any valid data e. g., the user enters all negative numbers for DAILYRAIN. The tutor uses a directed simulation because the objective is to make the student aware of the bug, not to teach program testing skills.

The strategy of directed simulation of values expands into four steps:

- First, the tutor informs the student that they are about to perform a hand simulation of code and identifies the code segment. In this instance, the tutor asks the student to hand simulate the main loop.
- Second, the tutor gives the student the values with which to perform the hand simulation. The tutor gives the student the values because the strategy selected is a directed simulation. In this instance the tutor asks the student to hand simulate the code when the user does not enter any legal values of DAILYRAIN.
- Third, the tutor asks the student to hand simulate the section of code with the provided values. In this instance the tutor asks the student to hand simulate the main loop with a set of values for DAILYRAIN that are all negative i. e., illegal.

- Fourth, the tutor identifies target variables (e. g., TOTALRAIN) and asks about the status of them. In this instance the tutor asks if the value of TOTALRAIN is legal after the hand simulation.

Figure 7.4 shows breakdowns into components of examples of the other three strategies, Justify, Explain, and Simplify. Each of the breakdowns of the steps follows the same pattern as the breakdown of the simulate values strategy shown in Figure 7.3. Each example in Figure 7.4 shows the breakdown of the strategy at one of the leaves under the nodes labelled Justify, Explain, and Simplify of the strategy tree in Figure 3.16. For example, the Justification strategy, which is used to achieve the objective of helping the student to generate a new plan for a goal, shows the steps in the strategy at the leftmost leaf in the tree in Figure 3.16.

In summary, this section identified prototypical, plausible analyses of the component steps of four major tutorial strategies. For example, the component steps of the strategy of asking the student to perform a hand simulation of a segment of code, with specified values, is displayed in Figure 7.3. The analyses were based on empirical data gathered in the Real Time Study and the Questionnaire Study. Future research will address more deeply the issue of the component steps of tutorial strategies.

OBJECTIVE : Make Student Aware : Bug
STRATEGY : Simulate Values : Directed
STRATEGY STEPS

1. Ask student to simulate code fragment
2. Provide values for simulation
3. Ask student to simulate and give values of variables
4. Ask status of target variables

Figure 7.3 : Steps of the Strategy "Simulate Values"

OBJECTIVE : Generate New Plan for Goal
STRATEGY : Justify : Existing Plan for Goal
STRATEGY STEPS

1. Identify goal
2. Ask student to identify plan
3. Ask student to justify existing plan for goal
4. Produce & critique new plan
   - 4.1 Produce new plan
   - 4.2 Critique new plan
   - 4.3 Repeat 4.1 & 4.2 until satisfactory new plan

OBJECTIVE : Correct Program Goal
STRATEGY : Explain Goal For Plan
STRATEGY STEPS

1. Identify plan
2. Ask student to identify goal
3. Ask student why used plan for goal
4. Correct goal
5. Produce & critique plan for goal

OBJECTIVE : Teach Student Concept : Interactive Loop
STRATEGY : Simplify : Read/Write Loop
STRATEGY STEPS

1. Name concept to teach
2. Identify simplification
3. Present simplification
4. Progressively increase complexity to target

Figure 7.4: Steps of Examples of Justify, Explain, and Simplify

7.2.2. Recovery From Failure

Tutorial plans do not always work perfectly. For example, suppose a student incorrectly attempts to read an entire file of single-digit numeric data with a single READ statement.[4] The tutor may decide on a simple verbal description of a loop with a READ statement in it as the strategy for helping the student to understand how a loop can be used to read numeric data from a file. However, when the tutor gives the student the simple verbal description, the student may not understand it. One reason could be that the student does not understand that a file of single-digit numeric data consists of single characters separated from one another by blanks and that a single READ statement only reads a single character in the file.

When the tutor gives the simple verbal description, the student fails to understand it. The tutor realizes that the failure occurred because the student does not know how 1) data are represented in a file and 2) they are scanned by a READ statement. To overcome the student's failure of understanding, and hence the tutorial planning failure, the tutor draws a picture of the file with the single-digit numeric data in it, draws a schematic loop with a single READ statement in it, and then hand simulates the loop, pointing with a finger to show how a single character in the file is scanned on each iteration of the loop. In short, the tutor planned to help the student in a way that led to a failure, and then recovered from the failure

[4] *This episode occurred in the Failure Recovery Study.*

with an another, more expressive, technique. The recovery strategy the tutor used is called *Object Mapping Simulation* because the tutor draws representations of the two main objects (the file and the loop) and then simulates the effects of the iterative READ statement on the file. Thus, the file and the loop are mapped to objects and then the actions of the loop on the data in the file are simulated.

The goal of the research on tutorial failures was not to build a working component of a tutoring system that could recover from tutoring failures: Many significantly difficult problems have to be solved to do so. Rather, the research on recovery from tutoring failure had the goal of developing an initial description of the knowledge that tutors have about 1) types of failures encountered during tutoring and 2) strategies that can be used to recover from the failures. Clearly, the knowledge about types of failures and recovery strategies is a prerequisite to constructing a failure-recovery module for a tutoring system for programming.

7.2.2.1. Types of Failures

The Failure Recovery Study provided the data on which this work is based. Tutors were observed while they helped novice programming students and *failure episodes* were identified. Failure episodes were defined as interchanges between the tutor and the student in which the tutor gave the student some information, the student failed to understand the information, the tutor recognized failure, and then took corrective action. Each failure episode was

characterized, the tutor's response to it was noted, and its effect on the student was recorded.

Tutorial failures arose primarily because the tutor made an incorrect assumption about the student. There were three very rough types of incorrect assumptions tutors made about the student:

• Ability Assumption Failure. The tutor attributes too much, or too little, ability to the student. The current treatment deals only with cases in which the tutor attributes too much ability to the student.

• Knowledge Assumption Failure. The tutor believes the student knows less, or more, than the student in fact knows. The current treatment deals only with cases in which the tutor attributes too much knowledge to the student.

• Intent Assumption Failure. The tutor believes that the student had one intention in writing a segment of code whereas, in fact, the student had a different intention.

This categorization of the types of assumption failures is very coarse grained because each type has many subtypes, which the analyses of the data did not permit to be fully enumerated. For example, in the first category of assumption failures, ability is treated as a simple three-valued variable with the values High, Medium, and Low. In reality, tutors make ability assumption failures about students' skills for testing programs; figuring out how to translate a verbal description of an algorithm into its counterpart in a programming language; etc. and not simply mistaken assumptions that e. g., the student is high ability when, in fact, the student is low ability. Nevertheless, the coarse grained categorization of

types of failures is useful for understanding how tutors identify and recover from failures.

7.2.2.2. Types of Recoveries

Tutors' recovery strategies were related to the type of failure which the recovery strategy was intended to remedy. For example, if the tutor made a Knowledge Assumption Failure, the tutor tried to provide the knowledge that the student needed; if the tutor misjudged the student's ability, then the tutor changed the difficulty of the student's task.

The preliminary analysis of recovery strategies focused on Ability Assumption Failure and Knowledge Assumption Failure. For each type of failure a common, concrete recovery strategy was identified and is described in this section: The recovery strategies tutors used to remedy Intent Assumption Failures were less clear cut than those used to remedy Ability and Knowledge Assumption Failures and therefore are not discussed here because they are a topic of future research.

Ability Assumption Failures were uncommon and seemed to occur when a tutor did not know the student very well. The recovery strategy used by tutors depended upon the direction in which they misjudged the student's ability. When they attributed too little intelligence to the student, tutors became more telegraphic about references and provided less connective material between points they wanted to make. When tutors attributed too much ability to the student they took the opposite tack and were more explicit about

references and provided more connective material between points they wanted to make. These strategies are not implemented in PEFR. When an Ability Assumption Failure occurs, PEFR merely says that it would give more or less information to the student.

Knowledge Assumption Failures arose primarily in three situations. Each of the three situations led to a different recovery strategy, which are depicted in Figure 7.5:

• Factual Information Strategy for Factual Knowledge Assumption Failure Sometimes a student simply "didn't know something" and the tutor told the student. For example, some student's did not know that division by zero causes a runtime error. When this occurred, tutors simply told students about the division by zero error and explained that such errors must be avoided because they cause the program to halt execution.

• Object Reification Strategy for Object Knowledge Assumption Failure Often a student might have a poor mental model of an object. For example, if a student did not know how single-digit numeric data are represented in a file, the tutor typically drew a picture of the file with single-digit numbers arranged on the lines of the files. Or, if the student did not understand Pascal records, the tutor might draw a large box to represent the record as a container and little boxes inside the large box to represent the fields of the record. Object Reification by drawing, or telling the student to visualize an object with certain properties, is intended to help the student build a mental model of an object that the tutor believes the student must be able to reason about for successful tutoring.

• Object Mapping Simulation Strategy for Object Relationship Assumption Failure A student might have a poor mental model of the relationship between two objects. For example, a student might not know how a WHILE loop with a READ statement iteratively reads data from a file of single-digit numbers. In this situation the tutor typically drew a picture of the two objects -- the file and the loop -- and then, using hand and finger motions, showed how the action of one object affected the other object. In the case of a student who did not know how a WHILE loop read data from a file of single-digit numbers, the tutor moved a finger on one hand over the numbers in the picture of the file while moving a finger on the other hand over each statement in the WHILE loop, stopping momentarily each time the READ statement was encountered and showing the student what happened to the variable into which the READ statement placed the number from the file.

The implementation of failure recovery strategies is very simple. PEFR does not determine the cause of the failure. The user of the program is asked for the cause of the failure e. g., Object Relationship Assumption Failure. PEFR uses the cause of the failure and the objects associated with the failure e. g., a file of numbers and a WHILE loop and performs table lookup to find the remedial strategy. When PEFR does not find a remedy, the user can add a remedy corresponding to the cause of the failure and the associated objects.

Failure	Strategy	Example
Factual Knowledge	Factual Information	Division by Zero Runtime Error
Object Knowledge	Object Reification	Pascal Records
Object Relationship	Object Mapping Sim.	Using WHILE loop to read from file

Figure 7.5: Knowledge Assumption Failures and Recoveries

7.3. Chapter Summary

This chapter has discussed two problems that arise when a tutorial plan is executed. First, tutorial strategies specified by TP must be broken down into component steps. For example, if TP specifies a strategy of directed simulation to make the student aware of a bug, the component steps of the strategy must be specified. For example, in one of the steps of the strategy of directed simulation the tutor must identify the code to simulate; in another step the tutor must tell the student which values to use for the simulation. The specification of the component steps of strategies is performed by PEFR, the tutorial plan execution module.

Second, when tutorial plans fail, the tutorial plan execution module must attempt to recover from the failure. Analyses of interactions of tutors and students revealed a coarse, three element categorization of tutorial failures. One of the types of failure,

Ability Assumption Failure, is remedied very simply. A second type of failure, Intention Assumption Failure, was not analyzed and reported here. The three chief kinds of Knowledge Assumption Failures, e. g., Object Knowledge Assumption Failure, were discussed and recovery strategies for them, such as Object Reification, were identified.

Chapter 8

GENERALITY OF APPROACH TO TUTORIAL PLANNING

8.1. Introduction

Domain generality is a major issue for any ITS, component of an ITS, or model of human tutorial reasoning. During the current, formative phase of building ITSs, it is only reasonable that progress toward the goal of powerful ITSs, and methods for building them, depend upon identifying the aspects of each existing ITS that are domain general, and therefore applicable beyond the specific ITS, and the aspects of the ITS that are domain specific and therefore not generally applicable. Doing so makes it possible to apply the lessons learned from building an ITS to future efforts. For example, analyses of CAI systems, which are based on a domain general, but rigid, control structure, and domain specific representations of knowledge, show that they do not provide much leverage for the goal of developing general approaches to machine based instruction.

The purpose of this chapter is to describe the aspects of TP which are not restricted to generating tutorial plans for the domain of novice programming. That is, the goal is to describe the main aspects of TP, and the knowledge on which TP is based, that are not tailored specifically to the domain of novice programming.

This chapter is divided into four major sections. The first section discusses the domain generality of the model of tutorial planning that is described in Chapter 2 and Chapter 3. The second section discusses that generality of TP, the implementation of the model of human tutorial planning. The third section identifies limitations of the model of tutorial planning; the fourth section provides some intuition about how the model described in this dissertation might apply to other domains.

8.2. Generality of Model of Tutorial Planning

This section discusses the domain generality of the four categories of knowledge that were described in Chapter 2 and Chapter 3. The first three types of knowledge are about students, the domain, and bugs. This knowledge is represented as records with slots and legal fillers for the slots. For example, one slot in the representation of the student is "ability", which has three possible values: "high", "medium" or "low". Knowledge about teaching falls into two categories: knowledge about tutorial planning and knowledge about tutorial plan execution.

8.2.1. Student

Students are represented with four main attributes each of which is domain general. The fillers for two of the slots -- ability and

motivation -- are also domain general. The fillers for the remaining two slots -- concepts and skills -- are domain specific.

The three-valued variables denoting the student's ability and motivation refer to the student's ability and motivation to work in the domain of the tutor, e..g., programming. The values of high, medium, and low, however, are independent of the domain of the tutor.

The remaining two attributes, denoting the student's concepts and skills, are domain general. A tutor for any domain would require knowledge about what the student knows and what the student can do. The values the attributes can assume, however, are not domain general. For example, iteration, one of the concepts a student may or may not understand, is a concept in the domain of programming. Iteration is not specific to Pascal programming, but it is not relevant to the domain of e. g., bacteriological diagnosis. The three values that denote the possible degrees of understanding a student has of a concept are domain general.

8.2.2. Domain

The tutor's knowledge of the domain of programming is obviously not general to all domains. The tutor's knowledge of the domain falls into four categories. The four categories and their limitations are:

- Task Specifications: As described in Chapter 2, task specifications describe the goals and subgoals for a programming assignment e.g., the Rainfall Task. Although the goals included in a particular task specification may be used in another task

specification (e.g., guard the update of a running total), specifications are specific to the domain of programming.

- Programming Plans: While the same programming plans may be used to solve several different programming tasks e.g., guarding an update of a variable is a common plan, programming plans are specific to the domain of computer programming.

- Concepts: Concepts such as iteration, sequencing, guarding, and so forth may have analogues in domains besides programming, but no claim is made that their representation in TP is domain general.

- Skills: Skills such as simulation, testing boundary conditions, and so forth may have analogues in domains besides programming, but no claim is made that their representation in TP is domain general.

Thus, although a case might be made that some of TP's knowledge about the domain of programming is not absolutely restricted to programming, no claim is made for its domain generality.

8.2.3. Bugs

Tutorial knowledge about bugs includes the four knowledge types of of causes, effects, plan component, and location. Knowledge of causes and knowledge of location are domain general whereas knowledge of effects and knowledge of plan component are not domain general.

The 13 causes of bugs that are used to generate a tutorial plans were identified in Chapter 2. With only four possible exceptions -- version hacking, try optimization, plan pollution, and failure to test -- the causes of bugs are not specific to the domain of programming. For example, slips are a common source of errors in many types of problem solving (Anderson, 1983). As well, misconceptions, misinterpreting a goal, and being uncertain about how to achieve a goal are common sources of errors in most domains of problem solving.

A plan for solving a goal is described by combinations of three location descriptors -- Before, Inside and After. Any domain in which the location of the components of a solution can be defined can use those three location descriptors. These there are two reasons that the three location descriptors, are not a complete set. First, the location descriptors refer to physical location of objects and not to e.g., temporal locations such as "Simultaneously".

Second, the three physical location descriptions required to generate tutorial plans for the programming domain refer to objects on a two dimensional plane. The three physical descriptors cannot handle all cases of placement of objects e.g., they are not sufficient for 3-space. For example, the location description "Behind", which might be required to describe the relative position of one component of a computer system with respect to another component, is not available in TP. However, the three location descriptions used by TP are a subset of location descriptions that would be required for a domain general spatial reasoning component of an ITS.

8.2.4. Generality of Tutorial Planning Decisions

The five Tutorial Planning Decisions that are the basis of the model of tutorial planning and TP, the implementation, are not restricted to the domain of novice computer programming. Regardless of the domain, each of the five Tutorial Planning Decisions must be made for a tutor to help a student with a multiple bugs.

Although the five Tutorial Planning Decisions are domain general, a few of the options for some of the five Tutorial Planning Decisions may not be. For example, hand simulation, one of the options in the teaching strategy taxonomy, is appropriate only in domains where there is an object to simulate. This section considers the taxonomy for each of the five Tutorial Planning Decision and identifies those options that may not be general across domains.

- Grouping: A grouping option identified in Figure 3.1 is applicable to any domain for which such a grouping is possible. None of the grouping options is inherently restricted to the domain of programming. Three kinds of groups, those based on bug dependencies, those based on a multi-component plan, and those based on plan components may have limited domain generality.

Some domains may not have an analogue of program behavior interactions, as defined in TP, because they do not have interacting components. For example, a solution to a problem consisting of a set of selections of e.g., stocks for an investor's portfolio, then the concept of "runtime interactions" is not sensible and therefore not applicable to that domain.

• Yes-No: The three possible options for whether to tutor a bug -- Definitely Yes; Try; Definitely Not; and If Opportunity Arises -- are not specific to the domain of computer programming.

• Ordering: It is plausible that the two primary ordering strategies shown in Figure 3.13 are not specific to the domain of programming. Although the other elements of the ordering taxonomy may not be universally applicable, they do not appear to be specific to the domain of programming.

• Objectives: The objectives listed under the node "Fix Program" in Figure 3.14 are not domain general. The three objectives of fixing the program code, the plans in the program, and the goals the student tried to solve in the program, are specific to programming. However, if the object of the three objectives -- the student's program -- were changed to be the object of the student's problem solving, then analogues of the three "Fix Program" objectives would be defined in virtually any domain.

For example, if the student were trying to design an automobile, then "Fix Program" would be changed to "Fix Design" and the programming domain objective of fixing the code would be changed to the design domain objective of fixing the student's design.

• Strategy: Most of the strategy options can be used in many domains. For example, asking for a justification, providing a simplified form of a problem for the student to solve, and explaining aspects of a solution are all domain general strategies. The only two strategies that are not obviously appropriate for all domains are 1) simulation (as discussed above) and asking, or

providing, an explanation for code. If the object of the explanation referred to entities in the domain analogous to code in the program domain, then it would be appropriate to ask for, or provide, an explanation of that entity. For example, a tutor for building architecture might ask a student who is designing an office building to explain the how the electrical wiring configuration supports the electrical power demands required by the building specifications.

8.2.5. Summary

The model of tutorial planning for multiple bugs in the domain of computer programming described in Chapters 2, 3, 4, and 5 is not inherently limited to the domain of programming. The 5 Tutorial Planning Decisions which must be made to generate a tutorial plan, and the four categories of knowledge on which the decisions depend are not specific to the domain of programming. For example, plan components, groupings of bugs, and teaching strategies are all concepts that have analogues in virtually any domain where the goal of problem solving is to assemble many components (e.g., mechanical parts; paragraphs; clusters of drugs) to achieve a solution.

Some types of knowledge in the four main categories are, of course, limited to the domain of programming. For example, many of the effects of bugs, such as "value is off-by-one" do not have analogues in other domains. On the other hand, it is unreasonable to suppose that the fact of domain specific types of knowledge would prevent the model of tutorial planning for multiple bugs from being applicable in other domains. For example, the concept of "value is

off by one" does not apply in the case of constructing a treatment plan for a patient with several illnesses that produce multiple signs and symptoms. The major work in the future, to increase the domain generality of the model, will be 1) to characterize the domains to which the types of knowledge in the four categories apply and 2) To identify additional knowledge types that are required for tutorial planning in additional domains.

8.3. Domain Generality of Implementation

This section addresses the extent to which TP's ability to construct tutorial plans is limited to the domain of novice computer programming. Because the the generality of TP depends upon its architecture and its tutorial planning rules, this section considers the domain generality of these two components.

8.3.1. Generality of Architecture of TP

TP's architecture, described in Chapter 5, is a straightforward, non-backtracking, rule-based design. TP's rules depend on declarative knowledge sources specific to the domain in which TP operates. For example, TP currently constructs tutorial plans for the domain of novice Pascal programming because its knowledge sources encode knowledge about the domain of novice Pascal programming.

To construct tutorial plans for another domain, TP's domain knowledge would have to be changed. If tutorial plans in the other

domain can be constructed by making the five Tutorial Planning Decisions, and TP's rules for making the decisions are adequate, then TP's architecture does not impose any limitation.

If generation of tutorial plans for the other domain necessitated heuristic evaluation of alternative plans and backtracking then TP in its current state could not construct adequate plans. Although heuristic evaluation of plans and dependency-based backtracking could be added to TP without compromising its basic tutorial planning goals or the rules used to achieve the goals, TP does not currently do so. Hence, TP's architecture does not currently handle selection of alternative tutorial plans but its form does not preclude the addition of a component to do so.

8.3.2. Generality of Tutorial Planning Rules

As Chapter 5 described, each of the five Tutorial Planning Decisions has a corresponding Tutorial Planning Goal in TP. In turn, TP achieves a tutorial planning goal by executing rules that are specific to that goal.

Tutorial planning rules are written in a domain general form. Each tutorial planning rule has two parts, a test part and an action part. The test part of a rule is constructed from a predicate base defined on e. g., the student, bugs, relationships among bugs, tutorial objectives, and tutorial strategies. The predicates for the test parts of the rules do not refer explicitly to the domain of programming. For example, one test predicate asks whether a bug is

is caused by a misconception. The answer to the predicate is either yes or no regardless of the domain in which the tutor operates. The test predicates are evaluated in the context of a knowledge base that is domain specific. If the knowledge base is for novice programming (as it is in the implementation of TP described in the dissertation) then the predicates are "about" novice programming. If the knowledge base is for the design of mechanical devices, such as automobiles, then the predicates are "about" automotive design.

The action base consists of operations performed on the tutorial plan that TP constructs. The actions that can be performed are not specific to the domain of programming. For example, one action is to record in the tutorial plan the decision not to tutor a particular bug. This action looks the same regardless of the domain in which TP operates. Other actions on tutorial plans are described below.

8.3.2.1. The Predicate Base

Figure 8.1 shows nine predicates in the predicate base. The nine predicates are sampled from the three types of knowledge described in Chapter 2. There are approximately 70 predicates in the predicate base for these three types of knowledge. For example, predicates defined on the student determine whether the student e.g., has high ability; knows a particular skill; knows a particular concept, etc. Predicates defined on bugs determine e.g., whether the effects of a bug are serious. Predicates defined on the domain (novice

programming in the case of TP) determine whether e.g., a plan is important or not.

Note that none of the predicates shown in Figure 8.1 contains any explicit references to the domain of novice programming. The predicates are designed to access data structures that contain domain specific information. For example, in the programming domain, each programming plan has a feature that encodes its importance. DOMPRED-ImportantPlan? accesses this information about the programming plan and returns "true" or "false". Any domain in which plans have an importance attribute, and in which the implementation of plan knowledge is consistent with the canonical form of plan knowledge in TP, automatically has DOMPRED-ImportantPlan? defined for it. The same is true of all the other predicates in TP.

8.3.2.2. The Action Base

Achieving a Tutorial Planning Goal alters the tutorial plan being constructed. Figure 8.2 shows most of the actions that TP takes to alter a tutorial plan. For example, the first Tutorial Planning Goal, Grouping, has three primary actions: make a new group, add a group to a plan, and add a bug to a group.

Each of the remaining four Tutorial Planning Goals is achieved by making corresponding alterations of the tutorial plan. For example, ordering the bug groups for tutoring can be viewed as constructing a specific permutation of the groups. TP uses two actions to construct a permutation of objects: Move one object

before another and move one object after another. Moving a group to the front of the tutorial plan is provided as a primitive operation, as is moving a group to the end of the tutorial plan.

K-Type	Example	Explanation
Student	STUDPRED=AbilityHigh? STUDPRED=KnowsSkill? STUDPRED=KnowsConcept?	Does the student have high ability? Does the student know the skill? Does the student know the concept?
Bugs	BUGPRED=EffectsSerious? BUGPRED=CauseIsSlip? BUGPRED=SamePlanComp?	Are the effects of the bug serious? Is the cause of the bug a slip? Are the bugs in the same plan component?
Domain	DOMPRED=ImportantPlan? DOMPRED=ImportantConcept? DOMPRED=TaskSubGoal?	Is the plan important? Is the concept important? Is the task a subgoal of the goal?

Figure 8.1: Examples of the Predicate Base

TP Goal	Example	Explanation
Grouping	ACTION=MakeNewGroup ACTION=AddGroupToPlan ACTION=AddToGroup	Create a new group of bugs Add a group to the tutorial plan Add bugs to existing group in tutorial plan
Yes-No	ACTION=MarkDoTutor ACTION=MarkNotTutor ACTION=MarkTutorIfTime	Mark a group to be tutored Mark a group to be left untutored Mark a group to be tutored if time permits
Order	ACTION=MoveBefore ACTION=MoveAfter ACTION=MoveToFront	Move a group in tutorial plan before another Move a group in tutorial plan after another Move group to front of tutorial plan
Objectives	ACTION=AddObjective ACTION=DeleteObjective	Add a tutorial objective to a group Delete a tutorial objective from a group
Strategy	ACTION=AddStrategy ACTION=DeleteStrategy	Add a tutorial strategy to a group Delete a tutorial strategy from a group

Figure 8.2 : Examples of the Actions TP Performs

8.3.3. Summary

The domain generality of TP, the implementation of the model of tutorial planning, depends upon 1) its architecture and 2) its tutorial planning rules. TP's architecture is a simple, forward chaining, rule-based design and therefore potentially general. Although TP does not perform heuristic evaluation of alternative tutorial plans, it would not be difficult to add such a process if a domain required it.

The rules that TP uses to generate tutorial plans are constructed from a base set of predicates and actions which are domain general. The rules that make the five Tutorial Planning Decisions are written in terms of the base set of predicates and actions. The reasoning about the five Tutorial Planning Decisions therefore occurs entirely at the level of these domain general predicates and actions. These predicates refer to domain knowledge to obtain their values: The rules for the five Tutorial Planning Decisions do not. The generalization of the tutorial planner to a new domain for which the current tutorial planning rules are adequate to construct tutorial plans thus entails specifying the domain knowledge on which the tutorial planning predicates and actions depend.

8.4. An Example from Another Domain

The purpose of this section is to give a brief illustration of the generality of the model of tutorial planning for multiple bugs. The argument will not show that the model is fully general. Rather, the argument will show that the analysis of tutorial planning for multiple bugs offered in this dissertation can be applied naturally to another domain which, on the surface, seems very unlike programming. Thus, rather than making it possible to conclude that the model of tutorial planning for multiple bugs applies to *all* domains, it will be possible to conclude that the model of tutorial planning is clearly *not restricted* to the domain of novice programming: This seems a reasonable first step in the addressing the research problem of the domain generality of the model.

The argument made in this section is focused on showing that the knowledge that is used to generate tutorial plans for multiple programming bugs can be used to generate tutorial plans for multiple bugs in another domain. Thus, the argument will illustrate how the five Tutorial Planning Decisions, and the knowledge that is required to make them, both apply to another domain.

The non-programming domain in which the applicability of the model of tutorial planning for multiple bugs will be illustrated is that of mechanical design, specifically the design of automobiles. The illustration will show the applicability of the five Tutorial Planning Decisions and their taxonomies to the task of tutoring a novice designer of automobiles. Clearly, an analysis of the domain

knowledge required to tutor the design of automobiles is far beyond the scope of the current work. Thus, the examples of bugs may seem simplistic. However, this is, in fact, a good sign because it shows that it is not necessary to look very deeply for evidence of the generality of the model of tutorial planning for multiple bugs.

The automobile design scenario is as follows: The student designer generates a potential design for an automobile. The design is in the form of pictures and text. The student's design has several bugs. Among the bugs are the following:

- BUG 1 - BUG 20: Lug nuts on wheel studs have round tops and cannot be fastened with standard lug wrenches.
- BUG 21 - BUG 24: Nuts on engine mounts have round tops and cannot be fastened with standard tools.
- BUG 25: Oil pump is placed next to path of electrical wires for computer sensor.
- BUG 26: Electrical wires for a computer sensor are wrapped with insulation against heat which causes temperature sensors to give false values.

A tutorial planner for the domain of car design would have to make the five Tutorial Planning Decisions to generate a tutorial plan for these 26 bugs. The tutorial plan is shown in Figure 8.3. The five Tutorial Planning Decisions, recapitulated in the figure, are made as follows:

Tutorial Planning Decision 1 Grouping, results in two bug groups.

Bug Group 1: BUG 1 to BUG 24. These 24 bugs are all in the same plan component, that of securing a device to the supporting

structure. The plan component is not focal. Thus, the bug group type is Same Plan Component Housekeeping.

Bug Group 2: BUG 25 and BUG 26. The second bug, BUG 26, wrapping the sensor wires in insulation, is a bad repair to the first bug, BUG 25, infelicitously placing the oil pump. Thus, the bug group type is Bad Repair.

Tutorial Planning Decision 2 The decision about whether or not to tutor, is made as follows: Bug Group 2 is the most important bug group and therefore must be tutored. Bug Group 1 may be addressed but only if there is time: it is far less important

Tutorial Planning Decisions 3 The ordering decision follows from 1) the second Tutorial Planning Decision, which determines the importance of the bug groups and 2) the fact that Bug Group 2 does not depend on tutoring any bug in Bug Group 1.

Tutorial Planning Decisions 4 The selection of objectives for Bug Group 1 is based on the effects of the bugs. The selection of objectives for Bug Group 2 derives from the same fact.

Tutorial Planning Decisions 5 The tutorial strategy selected for Bug Group 1 results from the objective of getting the student to see that the selected attachment method will not work. Detailed simulation of attaching or detaching the bolts with a wrench will get the student to see that the wrench cannot gain a purchase on the round bolt heads. The strategy for Bug Group 2, requesting justification of

the insulation of the wires, is intended to lead the student to label the insulation as a repair and to explain why it was selected. The tutor can then ask the student about the problem of disrupting an accurate signal in the temperature sensor wire.

In sum, the student's automobile design attempt contains many bugs. As the tutorial plan for the bugs shows, the five Tutorial Planning Decisions, and their associated taxonomies appear to apply naturally to this multi-bug tutoring situation. The example does not prove that the model of tutorial planning for multiple bugs is fully general but it does show how naturally the model might be extended to other domains. The immediate question is why this is so.

One reason may be the similarity of programs and automobiles. Both *are devices* in that they "run" and, in so doing, produce behavior. The second reason, closely related to the first, is that designing programs and automobiles requires mapping desired functions onto coordinated structures which, when operated, yield the desired function. Further research is required to characterize the domains to which the model of tutorial planning for multiple bugs can be naturally extended.

	GROUP 1: BUG1 TO BUG 24	GROUP 2: BUG 25 AND BUG
GROUPING	*Same Plan Component Housekeep* All the bugs in this group are in the same plan component (attach object with fastener) of different plans	*Bad Repair* The insulation on the wire se is to repair effect of heat of pump. It is a bad repair bec it disrupts temperature sens
YES/NO	IF TIME Easy to recognize and fix.	DEFINITELY This is a repair problem. Generating an acceptable alternative placement for the oil pump and an alternati path for wires will be hard.
ORDERING	SECOND Relatively easy to fix. Less important than repair bug	FIRST Important and may be hard to fix.
OBJECTIVE	DIFFERENT PLAN Want student to see that attachment method won't work.	DIFFERENT PLAN Placement not functional. Ma use alternative method of transmitting sensor informat
STRATEGY	REQUEST SIMULATION Want student to "run into" problem that was overlooked.	REQUEST JUSTIFICATION Want student first to see tha this is a repair. Then want student to see why this is a bad plan. Generating justifications is strategy for this.

Figure 8.3: The Five Tutorial Planning Decisions for Car Design Bugs

8.5. Chapter Summary

This chapter has described the main aspects of the knowledge required for tutorial planning, and the implementation of the tutorial planner, that are not specifically dependent on the domain of novice programming. The chapter showed that the four types of knowledge described in Chapter 2 and Chapter 3 are not specific to the domain of computer programming. For example, it was argued that the five Tutorial Planning Decisions must be made in order to construct a tutorial plan for multiple bugs in many domain.

In addition, the domain general aspects of TP, the implementation of the tutorial planner, were described. It was shown that both TP's architecture and processing mechanism do not restrict the generation of tutorial plans for multiple bugs to the domain of novice computer programming. The main limitation of TP's forward chaining, non-backtracking control structure is that it prevents TP from constructing several alternative tutorial plans and heuristically evaluating them to find the best. TP's architecture does not prohibit adding heuristic evaluation to the processing mechanism. If this type of processing is required to generate tutorial plans for either more complicated multiple bug programs or for different domains, TP's current form would make it straightforward to add such a mechanism.

Finally, an example was given to show that the model of tutorial planning for multiple bugs described in this dissertation extends naturally to other domains.

Chapter 9

CONCLUSIONS, LIMITATIONS, AND FUTURE DIRECTIONS

9.1. Introduction

This dissertation analyzed the problems that confront tutors when they try to help novice programming students who make more than one bug. The chief claim made in the dissertation is that effective tutors develop a *tutorial plan* to help a student with multiple bugs. This claim was supported by empirical analyses of tutorial planning strategies of experienced human tutors. The empirical analyses resulted in a model of the chief subtasks that must be accomplished to produce a tutorial plan and the rules that are used to carry out the subtasks.

The model of tutorial planning for multiple bugs is based on the five Tutorial Planning Decisions that must be made to produce a tutorial plan e.g., deciding whether or not to address each bug. A major contribution of the dissertation research is the identification of the knowledge required to make each of the five Tutorial Planning Decisions -- for example, a tutor typically reasons about the severity of a bug's effects in deciding whether or not to include it in the tutorial plan. The model of tutorial planning that was developed from the empirical analyses was implemented as TP, a computer

program that generates tutorial plans that are similar to the ones that experienced human tutors generate.

The remainder of this chapter addresses major limitations of the present state of the model of tutorial planning for multiple bugs. First, three limitations resulting from the focus of the dissertation on the most prevalent multiple bug tutoring situations are identified -- for each of these limitations, directions for immediate research are described. Second, less immediate, but nonetheless important, issues that that have emerged during the research are considered.

9.2. Limitations and Immediate Future Directions

The major goal of the research reported in the dissertation was to account for the tutorial planning that occurs in the multiple bug situations with which tutors are most often confronted. As a result of the focus, certain issues that are important were neglected but not forgotten. The purpose of this section is to identify several of the chief limitations and, for each, to indicate how it will be addressed in the next phase of research.

• Limitation 1: *Small Number of Programming Tasks.* TP currently can develop tutorial plans for two task specifications. Each task specification is represented as a a graph of goals to satisfy. TP currently has a knowledge base of approximately 40 reusable goals. The next phase of research will make TP's goal specification knowledge base more extensive so that more task specifications can be represented.

• Limitation 2: *Weak Tutorial Plan Execution & Failure Recovery.* TP's strength lies in its ability to construct tutorial plans. TP does not have a powerful tutorial plan executor nor does it have many strategies for recovering from failures that occur during the execution of its tutorial plans. The next phase of research on tutorial planning will be to study, in detail, how experienced human tutors execute tutorial plans and how they cope with failures that occur during tutoring.

Studies of tutorial plan execution will focus on techniques that tutors use to deliver instruction to students. For example, a tutorial plan may specify that the tutor use the strategy of asking the student a "why" question. In this case a human tutor would use expectations about what the student should say to monitor the answer. These expectations are used to determine whether the information the student gives is 1) correct and 2) sufficient. The next phase of research will identify the form of these expectations for the primary strategies that tutors specify in their tutorial plans.

Studies of failure recovery will attempt to determine the conditions under which tutors use a particular recovery strategy. Preliminary empirical studies of human tutors' recovery failure strategies, reported in Chapter 7, suggest that tutors use a relatively small set of techniques for recovering from failure. For example, a tutorial failure may occur if a student does not understand the action of an input loop that reads from file, and the tutor's plan was based on the assumption that the student did understand it. If a failure does occur then the tutor may resort to drawing a picture of the file, writing a loop with a read statement,

and hand simulating the action of the loop on the file, illustrating how the file pointer moves with each iteration of the loop. The strategy of drawing pictures to make abstract concepts more concrete is a very common strategy that tutors use to recover from certain types of failures. The next phase of research will identify these strategies and the conditions in which they are used.

• Limitation 3: *Difficult Tutoring Situations.* TP currently can generate acceptable tutorial plans for approximately 65% of the multiple bug situations with which it is confronted. TP's knowledge about how to plan for multiple bugs is based on the approximately 13 common groups of multiple bugs. When TP is confronted with bugs that do not fit neatly into known groupings, it is unable to construct a tutorial plan that is as acceptable as those it builds when the grouping is known. One of the causes of this limitation is that TP does not construct and evaluate alternative plans for the same bugs: Human tutors do appear to construct and evaluate alternative tutorial plans when they are confronted with unfamiliar multiple bug situations.

In the next phase of research on tutorial planning, human tutors will be studied when they must construct tutorial plans for unfamiliar, or difficult, tutoring situations where there is not one clearly acceptable plan. The studies of human tutors will identify the generation and evaluation strategies that result in the of an acceptable plan. These generate and test strategies will then be incorporated in the next version of TP.

In summary, the current state of understanding of tutorial planning for multiple bugs is limited in, at least, three important

ways. The first limitation, the small number of programming tasks TP understands, is not serious and it will be straightforward to remedy it. The remaining two limitations -- related to tutorial plan execution and difficult tutorial planning tasks -- are deeper than the first. The approach to the two more serious limitations will be to perform empirical studies of the behavior of experienced human tutors and to augment both the model and the implementation of tutorial planning based on the findings from the empirical studies.

9.3. Additional Issues

During the course of the research on tutorial planning, several topics have arisen that do not qualify for immediate treatment but, rather, are general issues that should be kept in mind when doing the next phases of the research. The purpose of this section is to identify four such topics.

First, the model of tutorial planning described in the dissertation is not intended to be a process model of the tutorial planning of experienced human tutors. Rather, the model described is a descriptive model. That is, the model identifies the knowledge that human tutors use but does not give a detailed account of the control mechanism that guides the use of the knowledge. Empirical studies of the process of tutorial planning, analogous to the studies of tutorial plan execution and failure recovery, should be be performed to clarify the process of tutorial planning.

Second, acquisition of expertise and individual differences among tutors should be studied. Effective tutors are not born, but rather learn to become good tutors. It is important to determine how to teach tutors to become experts.

Third, it is necessary to keep in mind the goal of constructing a full tutoring system for novice programming. The tutorial planner is only one component of a much larger system including a bug finder, a student modeler, a domain expert, etc. The problem of integrating the tutorial planner into a complete tutoring system should be kept in mind as studies of the control structure of human tutors are designed. It seems very plausible to suppose that careful empirical study of the control structures that integrate the tasks of bug finding, bug repair, tutorial planning and tutoring of experienced human tutors will provide valuable information for integrating these tasks in a machine tutor.

Finally, experienced human tutors have naive, or *volkpsychologie*, models of human learning: Such naive models may or may not be the same as the "correct" models derived from controlled empirical studies of learning. Nonetheless, if one goal is to teach tutors to become better tutors -- and that is a goal of this research -- then it is necessary 1) to describe the naive models of learning held by tutors and 2) to identify the aspects of the models that are correct and, thus, should be left intact, and those aspects of the models that are incorrect and, thus, should be a focus of efforts to improve the expertise of human tutors. For example, if an ITS for tutorial planning were to use the plan critiquing methodology of Miller, 1986, it would be necessary to understand the naive models

of learning held by tutors so that their inaccurate models could be transformed to more accurate models.

9.4. Final Summary

This dissertation has provided an empirically based analysis of *tutorial planning* by human tutors. The main result is the identification of the knowledge required to generate tutorial plans for multiple bugs that have the same form as, and are therefore as acceptable as, tutorial plans for multiple bugs generated by experienced human tutors. Three limitations of the research were identified. Each limitation leads directly to a topic of future research. Finally, several more general issues that are entailed by any research on tutoring systems for complex problem solving were identified and their importance briefly noted.

REFERENCES

Anderson, J, The architecture of cognition. Harvard University Press, Cambridge, MA, 1983.

Anderson, J., Boyle, C., Farrell, R., and Reiser, B. Cognitive principles in the design of computer tutors. Technical Report, Advanced Computer Tutoring Project, Carnegie-Mellon University, 1984.

Brown, J. S. and Burton, R. Diagnostic models for procedural bugs in basic mathematical skills. Cognitive Science, 2, 155-192, 1978.

R. Burton and Brown, J. S. An investigation of computer coaching for informal learning activities. In Intelligent tutoring systems. D. Sleeman and J. S. Brown (eds.), Academic Press, London, 1982.

Carbonell, J. AI in CAI: An artificial intelligence approach to computer assisted instruction. IEEE Transactions on Man-Machine Systems, MMS-M, 4, 1970.

Clancey, W. Guidon. Journal of computer-based instruction, Summer 1983, Vol. 10, Nos. 1 & 2, 8 - 15.

Collins, A. and Stevens, A. Goals and strategies of interactive teachers. Technical Report #3518, 1976 Bolt, Beranek, and Newman, Cambridge, MA.

Collins, A. Strategies of human tutors. Paper presented at ONR workshop, University of Illinois, July, 1986.

Dixon, W. & Massey, F. Introduction to statistical analysis. McGraw-Hill, New York, NY, 1957.

Johnson, L., Soloway, E., Cutler, B., and Draper, S. Bug catalogue: I. Technical Report #286, 1983, Department of Computer Science, Yale University, New Haven CT.

Littman, D., Pinto, J., Soloway, E. Observations on tutorial expertise. Proceedings of IEEE Conference on Expert Systems in Government, Washington, D.C. 1985.

Littman, D., Pinto, J., Soloway, E. An analysis of tutorial reasoning about programming bugs. In *Proceedings of 1986 National Conference on Artificial Intelligence* (AAAI86).

Littman, D. & Soloway, E. Toward an empirically-based process model for a machine programming tutor. In *Proceedings of IEEE Conference on Systems, Man and Cybernetics,* Atlanta, October 1986.

Littman, D., Pinto, J., & Soloway, E. The knowledge required for tutorial planning: An empirical analysis. Paper presented at Third Annual Conference of Artificial Intelligence And Education, Pittsburgh, PA, May, 1987.

Littman, D. Tutorial planning in complex tutoring settings. Invited paper presented at Third CERCLE Workshop on Teaching Knowledge And Intelligent Tutoring, Lancaster, England, April, 1988; to be published in edited volume of proceedings.

Littman, D., Pinto, J., & Soloway, E. The knowledge required for tutorial planning: An empirical analysis. Paper in preparation, 1989.

Miller, P. Goal-directed critiquing systems: Practice-based medical consultation by computer. Springer-Verlag, NY, 1986.

Newell, A. and Simon, H. Human problem solving. Prentice-Hall Englewood Cliffs, NJ, 1972.

Peachy, D. & McCalla, G. Using planning techniques in intelligent tutoring systems. International Journal of Man-Machine Studies, 24, 77-98, 1986.

Perkins, D. and Martin, F. Fragile knowledge and neglected strategies in novice programmers. In: E. Soloway and S. Iyengar (eds.) Empirical studies of programmers, Ablex Publishers, 1986. Also republished in Journal Of Systems and Software, 1987.

Soloway, E., Woolf, B., Barth, P., & Rubin, E. MENO-II: An intelligent tutoring system for novice programmers. Proceedings of the

Seventh International Joint Conference on Artificial Intelligence, Vancouver, Canada, 1981.

Spohrer, J., Pope, E., Lipman, M., Sack, W., Freiman, S., Littman, D., Johnson, L., Soloway, E. Bug catalogue: II, III, IV. Technical Report, Department of Computer Science, Yale University, 1985.

Spohrer, J. and Soloway, E. Analyzing the high-frequency bugs in novice programs. To appear in: Workshop on empirical studies of programmers, E. Soloway and S. Iyengar (eds.), Ablex, Inc., 1986.

Spohrer, J. MARCEL: A GTD Impasse/Repair Model of Student Program Generation, Doctoral Dissertation, Department of Computer Science, Yale University, New Haven, CT, 1989.

Stevens, A., Collins, A., and Goldin, S. Misconceptions in students' understanding. In Intelligent tutoring systems. D. Sleeman and J. S. Brown (eds.), Academic Press, London, 1982.

Woolf, B. Context dependent planning in a machine tutor. Doctoral Dissertation, University of Massachusetts, Amherst, MA 1984.

Appendix I

Pre-Dissertation Work on Multiple Programming Bugs

This appendix discusses some preliminary work on the problems that arise when trying to decide what, and how, to tell students about multiple bugs in their programs. The discussion first describes the approach to reporting bugs taken by PROUST (Johnson, 1985) and some of the limitations of PROUST's bug reports. Following identification of some of the main problems of PROUST's bug report, a brief discussion is given of some initial work on reporting multiple bugs. The initial work resulted in 1) minor modifications to PROUST that provided students with additional information about their bugs and 2) a clearer idea of the issues entailed by multiple bugs. This work thus set the stage for the dissertation work.

PROUST is a computer program written by Lewis Johnson (Johnson, 1975) that finds non-syntactic bugs in novice programmers' PASCAL programs. PROUST takes as input a student's syntactically correct program and produces as output a listing of the bugs in the program. For example, Figure 1.1 shows a student's program that has nine bugs. When PROUST reports bugs to a student it lists the bugs in textual order and gives the line number on which the bug was found. Thus, PROUST's output is not organized for tutoring. The output is intended simply to make student's aware that bugs are present and to suggest some possible repairs.

PROUST's identification of bugs is useful to students (Sack, Littman, Soloway, 1987) but it cannot be considered optimal for several reasons:

- PROUST reports all the bugs to the student. For example, in the program shown in Figure 1.1, even though BUG 8 and BUG 9 are bugs, they are so unimportant that most tutors do not even tell students about them unless the rest of their programs are perfect because they do not want to burden the student with too much to think about. PROUST not only tells students about these bugs, it does not tell students that they are "unimportant".

- PROUST reports bugs in textual order. Experienced tutors try to work with students on common threads that run through several bugs which may not be adjacent. In Figure 1.1, for example, most tutors would begin the tutoring session by eliminating the infinite loop caused by BUG 4. The infinite loop is both a hard bug to fix and it is the symptom that brought the student to the tutor in the first place.

- PROUST does not detect and group together "similar" bugs. For example, a human tutor who decided to address BUG 8 and BUG 9 would note the similarity of the bugs, treat them together, and probably tell the student about the similarity: PROUST does not tell students about bugs that have a common theme.

- PROUST does not tell the student which bugs are important. As with the problem of telling students about all bugs, PROUST's output does not give students information about which bugs are important and which bugs are minor.

- PROUST does not help the student figure out which bugs to correct first. PROUST cannot tell students what they should try to fix first. The first step in correcting the program in Figure 1.1 would be to eliminate the infinite loop. Not all students have this piece of knowledge and PROUST does not provide it.

Thus, while PROUST is very good at its original mission, finding bugs, it is not sophisticated about helping students with them. In particular, PROUST does not give students useful information about what bugs are important.

To overcome some of the limitations of PROUST's reports to students, I added a program module to PROUST which organized its output to be more useful to the student. In this first attempt to organize bug reports for students, the main goals were to tell students what bugs were important and to identify each bug with a large organizing segment of the program, such as the main processing loop, or the output section.

The modified PROUST identified bugs for students as either critical or minor and told students which segment of the program each bug was associated with. For example, Figure MODPROUST1 shows some of the output of PROUST that a student might see for the program in Figure 1.1. Notice first that the bugs are reported according to a natural segmentation of the program into major sections -- the fragment of output shown in the figure reports bugs from the main processing loop. Second, note that the bugs that are critical are reported first to the student and are labelled as such.

Although the addition of rules to organize PROUST's output of bugs seemed to make it more useful to students, it fell far short of

producing the organization of the bugs that a human tutor would use. Chief among the remaining problems with the output were:

• The modified output did not suggest to the student relationships among bugs. A trivial example from the program in Figure 1.1 is that the modified PROUST would not realize that BUG 8 and BUG 9 were essentially "the same bug" and therefore could not tell the student about the single input case in which both bugs arise.

• PROUST still told the student about each bug, regardless of its importance.

• PROUST could not tell students about bug dependencies. For example, students often make one bug, detect it, and create a second bug in the attempt to repair the first. Human tutors almost always treat such patterns of bugs together. PROUST cannot.

Thus, although the modified output of PROUST gave students additional information that may have been helpful, it was not a satisfactory solution to the problem of multiple bugs in the programming domain and it certainly did not provide a solution that could be useful in domains other than programming.